AFIT/GCS/ENG/92D-15

ENTITY-RELATIONSHIP VERSUS OBJECT-ORIENTED MODELING AND THE UNDERLYING DBMS

THESIS

Kevin Joseph Routhier
Captain, USAF

AFIT/GCS/ENG/92D-15

Approved for public release; distribution unlimited

AFIT/GCS/ENG/92D-15

ENTITY-RELATIONSHIP VERSUS OBJECT-ORIENTED MODELING AND THE UNDERLYING DBMS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Systems)

Kevin Joseph Routhier, B.S.E.E.

Captain, USAF

December 1992

Approved for public release; distribution unlimited

Preface

"The Lord gives strength to his people; the Lord blesses his people with peace." [30:(Psalm 29:11)]

First of all I must give praise and thanks to my Lord and Savior, Jesus the Christ. Through His strength and peace I was able to work hard and learn a great deal while actually enjoying myself, my family, and friends during both this thesis effort and my entire AFIT assignment. "Thanks be to God for his indescribable gift" [30:(2 Corinthians 9:15)]!

To continue my thanks in order of importance, I must next lift up my wife, Wendy, whose all-encompassing support from the home-front made it possible for me to do my job well. To my two sons, Caleb and Josiah, I thank you for the hugs, kisses, and smiles that greeted me each day, though our play time together was sometimes *shorter* than you would have liked. To my beautiful new daughter, Jordan, who is proof that there is more to AFIT than studying and late-night projects, I thank you for being born at such a strategic time. Along with my family, I would like to thank the many friends I have made at AFIT and in "the church." Their ongoing fellowship and encouragement made my time at AFIT most enjoyable.

Finally, special thanks with respect to the physical thesis goes to my thesis committee: Captain Dawn A. Guido, my thesis advisor; and readers, Lieutenant Colonel Patricia K. Lawlis and Major Mark A. Roth. Their guidance and encouragement allowed me to go where the research led, and in the process learn much about a subject I initially knew nothing about.

Additional thanks to the many instructors who indirectly assisted in my thesis through the knowledge gained from their classes.

As research for Department of Defense decisions to move in the object-oriented direction, this thesis may be a little late, since some organizations have already decided to move in that direction. For example Air Force Standard Systems Center has decided that all development will essentially be object-oriented as part of their modernization program [52]. Hopefully this thesis will, however, help to support such decisions and encourage further acceptance of the object-oriented paradigm throughout the military.

Note there is a glossary of technical terms located in Appendix A.

Kevin J. Routhier

Table of Contents

List of Figures

List of Tables

Abstract

Despite the impressive accomplishments in relational database research, greater support is needed for persistence of the new types of data encountered with object-oriented programming. The concept of object-orientation is not new in the realm of programming; however, its utilization in database management systems is still immature. Regardless of this fact, there is an urgency for object-oriented database technology.

With this increase in demand for the next generation databases comes the need to examine object-oriented data modeling versus the conventional entity-relationship modeling of relational database design. This thesis objective is to analyze both paradigms to determine if object-oriented modeling can significantly improve Department of Defense systems. After analyzing the entity-relationship paradigm and a representation of object-oriented modeling techniques we see a unifying of conceptual models encompassing both application and database development. Object-orientation's higher level of abstraction enables modeling of all problem domains and provides a common language between developer and client.

The critical issue in adoption of the object-oriented paradigm becomes not whether to adopt, but how to adopt object-oriented techniques. The benefits of object-oriented technology close the semantic gap by helping the computer to "see" things our way.

ENTITY-RELATIONSHIP VERSUS OBJECT-ORIENTED MODELING AND THE UNDERLYING DBMS

I. Introduction

1.1 Overview

Object-oriented database management systems (OODBMS) can mean different things to different people. Many object-oriented databases are currently being marketed and used, although some object-oriented researchers would not consider them "truly" object-oriented. The research into this next generation database system is dynamic and buzzing with controversy over terminology and exactly what object-oriented databases should contain and do. "Three points characterize the field at this stage: (i) the lack of a common data model, (ii) the lack of formal foundations and (iii) strong experimental activity" [1:2]. However, one thing is sure in the database industry today: if the term "object-oriented" is in the name or title, a product or book sells! When referring to the current OODBMS research efforts to solve the requirements of a completely object-oriented database management system (DBMS), Cattell asserts that, "in the DBMS marketplace, customers will not be willing to wait for one DBMS that does everything" [10:3]. It appears that there is an urgency for OODBMS technology even though the area is not fully realized.

This research will concentrate on the data modeling area by comparing the most recent object-oriented (OO) information models with

the contemporary entity-relationship (ER) model(s) while referencing their respective database management systems. OO and ER paradigms are both methods for representing software data requirements and for conceptual database design. Both are concerned with modeling real-world objects or entities, their attributes, and their relationships to each other. OO, however, goes beyond the ER paradigm by associating behavior with the objects in addition to their data structure. Moving from ER to OO data modeling changes the traditional view of programs and programming from a collection of functions performed on data to a collection of abstract objects defined by their functionality or services. It is characteristic that such a modeling approach can be complex and diverse, and be resisted by more traditional designers. Codd states that until a comprehensive data model is accepted for the object-oriented approach, investment in OODBMS is risky [14:480]. While the risk of embracing the object-oriented database management system is controversial, the prevalence of data modeling techniques for object-oriented design remain undisputed. Dyer and Roth contend that object-oriented design methods are beneficial regardless of the database management system selected, and can be used even when the risk of using OODBMS is deemed extensive [20:1-2].

1.2 Background

Before the OODBMS, two generations of database systems were prevalent in the 1970's and 1980's, respectively; thus the name third-generation has been given the object-oriented database system of the 1990's [48:1]. Figure 1 illustrates that the application and database environments have gradually transferred application program responsibilities to the

database management arena. The desired capabilities of the third-generation database system appear to push even further in the same direction as behavior enters the DBMS environment to supplant functions. This characterizes a shift from the concept of performing functions on data to the concept of discrete objects that contain both identifying data and behavior. It is important to note that as database exploration continues in this direction, the major advances realized from prior database generations "must not be compromised by third-generation systems" [48:4].

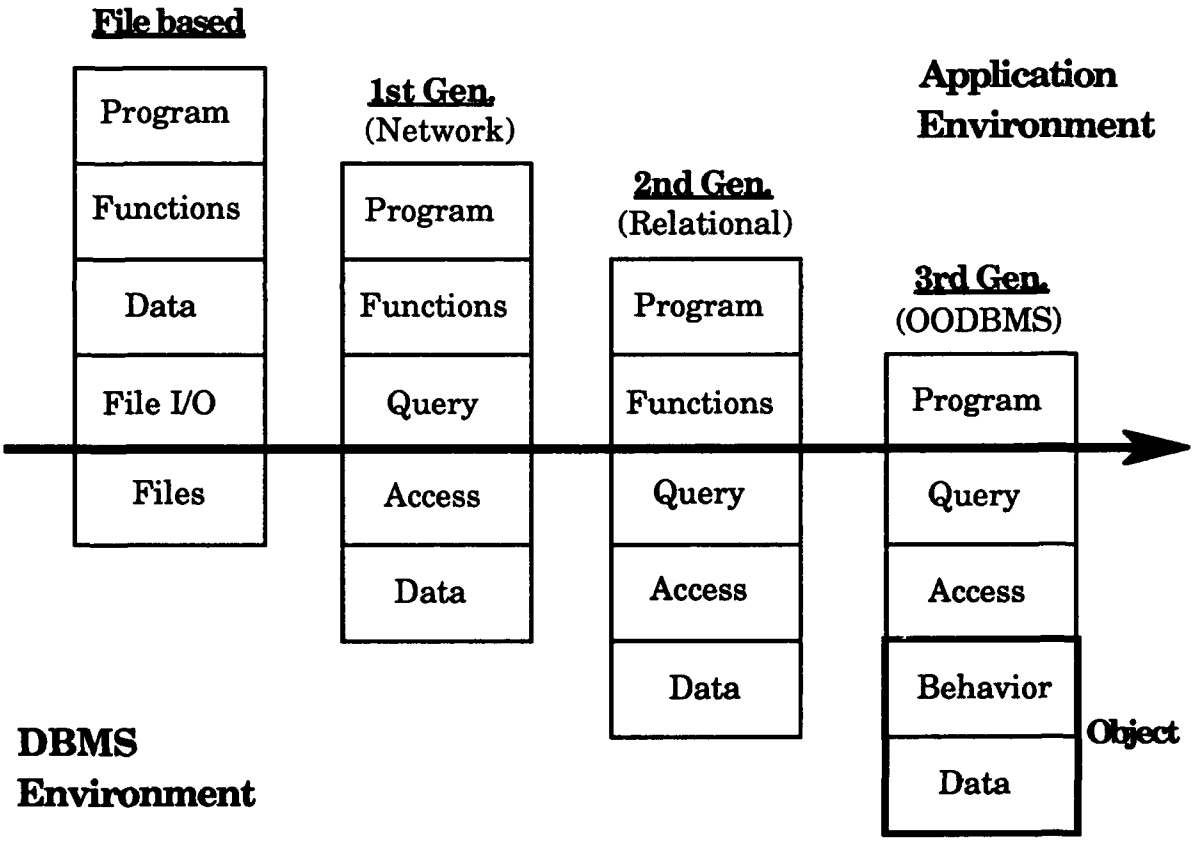


Figure 1. Evolution of Database and Application Environments (adapted from [38:5]).

File-based systems often require their data to be structured specifically for the particular application program using the data. This dependence resulted in considerable duplication of data and a laborious time keeping this data consistent among the many files. When the first-generation databases came into existence, this once redundant and application dependent data was kept regulated in one uniform "file" or database. Management of access to the data became the responsibility of the database management system, so data integrity could be achieved [53:2]. Another benefit was one data manipulation language could be used to collect data records. However, navigation through the database network or hierarchy had to be accomplished by the application programmer. "As a result, the database systems of 1970 were costly to use because of the low-level interface between the application program and the DBMS, and because the dynamic nature of user data mandates continued program maintenance" [47:112].

The second-generation relational databases took away the navigation complexity problem inherent in the network databases by structuring the data in an easy-to-understand tabular form. The addition of a high-level query language into the relational database system reduced the amount of code required by the application programmer to get to the needed data. Consequently, data independence was realized, freeing the user from the need to know or care how the data was physically or logically stored. These past two decades of technological advances have helped to close the semantic gap between the application programmer's "real-world view" and the logical schema of the database and its physical schema implementation as shown in Figure 2. "These capabilities freed the user from dealing with

low-level details of physical data organization, recovery, and coordination among users" [10:49].

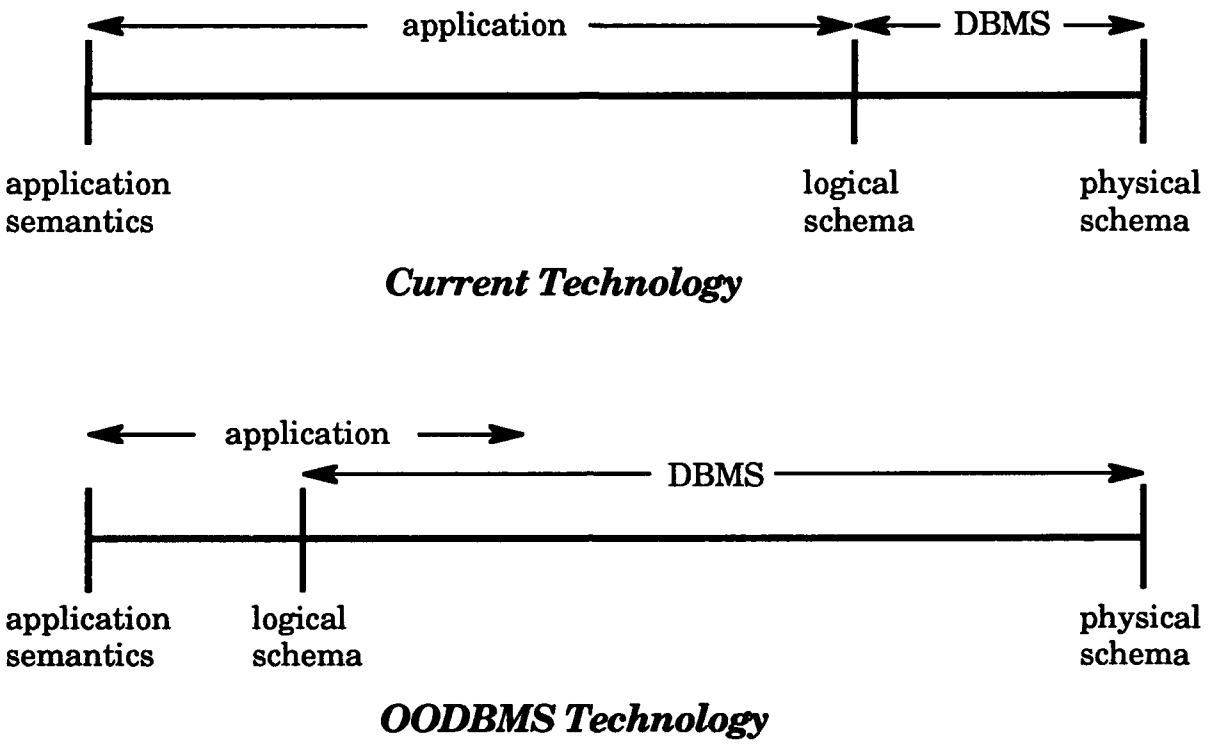


Figure 2. Closing the Semantic Gap [38:23].

The third generation of OODBMS technology promises to close this semantic gap even further because it "has user-oriented and developer-oriented concepts" [38:22]. These OODBMS concepts appear promising considering the impressive accomplishments to date. There are, however, many legitimate concerns about performance and reliability inherent with any newly proposed technology.

Throughout the history of DBMS technology, information modeling has taken on many forms to describe the data stored in databases, show any required consistency constraints, establish a logical schema and graphically represent how the data is related. The current entity-relationship models are being modified or replaced to provide new conceptual tools for the evolving OODBMS. Since the OODBMS will close the semantic gap by storing more "real-world" objects which may require handling far more complex data than the relational database, the resulting data model(s) will likewise be more convoluted [10:5].

1.3 Problem Statement

As programming evolved toward the object-oriented paradigm, resulting in the need for a supporting object-oriented database management system, many techniques for modeling object-oriented information emerged. However, no standard information modeling paradigm has emerged to assist in the design of object-oriented databases. "Most object-oriented analysis and design techniques are informal and are prone to ambiguity and inconsistency" [15:18]. "No comprehensive data model has yet been published for the object-oriented approach. To be comprehensive, it must support all of the well-known requirements of database management" [14:480]. An examination of object-oriented information modeling compared to traditional entity-relationship modeling and the underlying database management systems is needed to address the risk of investing in the object-oriented approach.

1.4 Research Objectives

The overall purpose of this research is to discover whether the object-oriented software technology can significantly improve Department of Defense systems. This project, sponsored by Software Technology for Adaptable Reliable Systems (STARS), is to compare the entity relationship and object-oriented information modeling techniques and, accordingly, the underlying relational DBMS and OODBMS to see if object-orientation is the paradigm to adopt in software engineering. The research will reflect current modeling techniques and their potential for software and database design, and subsequently, feasible and actualized object-oriented technology benefits, touching on areas of interest to military applications, maturity, standards, distributed systems, portability, reuse, embedded system potential, etc.

1.5 Summary of Current Knowledge

1.5.1 Object-Oriented Basics. The terminology may differ among resources, but the basic concepts of object-oriented software remain much the same. We begin with an object, which is an abstract entity or anything we wish to model in our system. The object has attributes or instance variables that contain data about the object together with behavioral characteristics called services or methods. The object attributes can themselves be objects. This concept can be easily understood when we think of an assembly line robot as an object. The robot can provide certain services and has attributes like sensors, arms, claws, etc. Each of these attributes can be an object with capabilities and attributes of its own. This

abstraction concept of separating behavior from implementation is much like the abstract data type (ADT) found in programming. Our robot object is much like a stack ADT that has a data structure and operations (push, pop, top, is empty, is full) that can be performed on it, but at the macro level. Because of encapsulation, code which uses the ADT is not affected by changes in the implementation of its behavior [31:62].

Objects have interfaces with the rest of the system which consists of a set of messages [35:428] or service requests [15:9]. These messages are simply requests passed to objects or among objects without them worrying about how the request is accomplished. Each object has implementation code, called methods or services, through which it responds to the messages. The method could be as simple as returning a constant value or some elaborate function to calculate a value or perform some operation. Our robot, for example, could be sent a message by the assembly line manager to paint the next car red, and it will do it without further painting instructions needed. All it needs to know is what color to paint the car, since it is already programmed (has a method) for how to paint the car.

Just as global data in application programs evolved toward the localization of variables into procedures (see Figure 3), so the network and relational database information—which is essentially global data—is localized with the methods into the objects of the object-oriented database (see Figure 4). When variables became localized their access became controlled through parameters (note the added parentheses in Figure 3). Similarly, the access to object data values is controlled through its services. The modularity, maintenance and reuse benefits obtained through

localization, and lost again with the introduction of the database, is once again restored with the object-oriented database management system. The result is an "intelligent" database the application programmer can use or direct to complete the mission without worrying how the database accomplishes its tasks. [44]

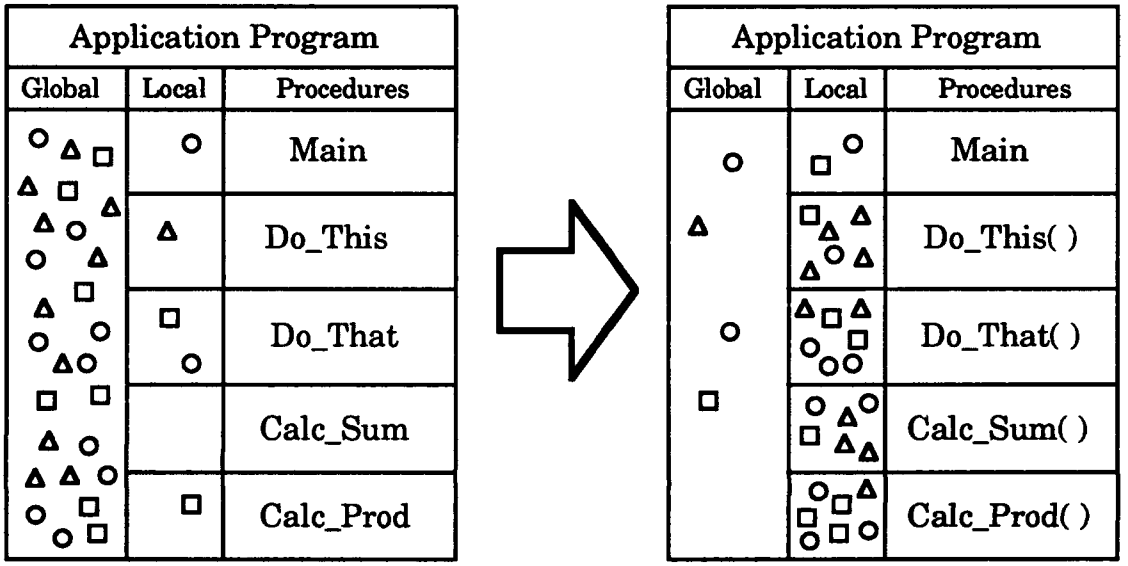


Figure 3. Localization of Variables.

The object-oriented database stores complex data types and the behavior rules (services or methods) that need to go with them. An object may change the way it is physically stored depending on the number of object instantiations. To be efficient, the storage may go from an array, to linked list, to a hash or B-tree as it grows in size, without the application noticing. This ability to modify how an object is defined without affecting the rest of the system is called data independence.

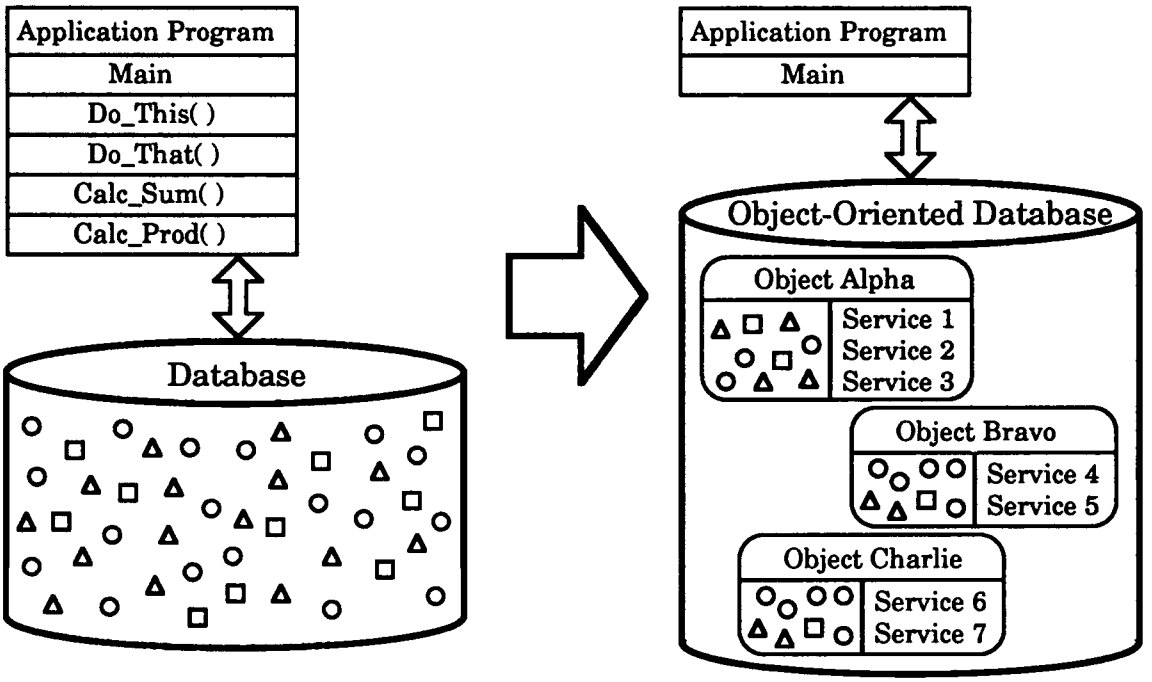


Figure 4. Localization of Object Data.

Groups of similar objects become a class, where the objects are instantiations of the class. Polygon is an example class with a possibly infinite number of instances of polygon objects that embody the class. Objects are similar when they respond to the same messages, use the same methods, and have attributes of the same name and type [35:428]. When classes have similarities, they can be placed into hierarchies to allow inheritance of shared-valued variables, as well as shared methods. "Inheritance is a relationship between classes which allows a new class to be defined as the specialization of another, its parent" [15:10]. This class-superclass relationship is very similar to an "ISA" or "IS-A" relationship in an entity-relationship model, where lower-level entity sets inherit properties from the higher-level entity set(s). Changes can be made

instantly to all subclasses through inheritance because code reuse is intrinsic to the system. Consequently, the possibility of huge ripple effects can arise for maintenance if the classes are not structured well. The likelihood of multiple inheritance exists if a class inherits characteristics from two or more superclasses. Any conflict with inherited, same-named properties (as shown in Figure 5) is handled by giving precedence to the locally defined property. At any time the user can explicitly choose to override a default by redefining the service [15:10] or specifying which superclass property to inherit [33:28].

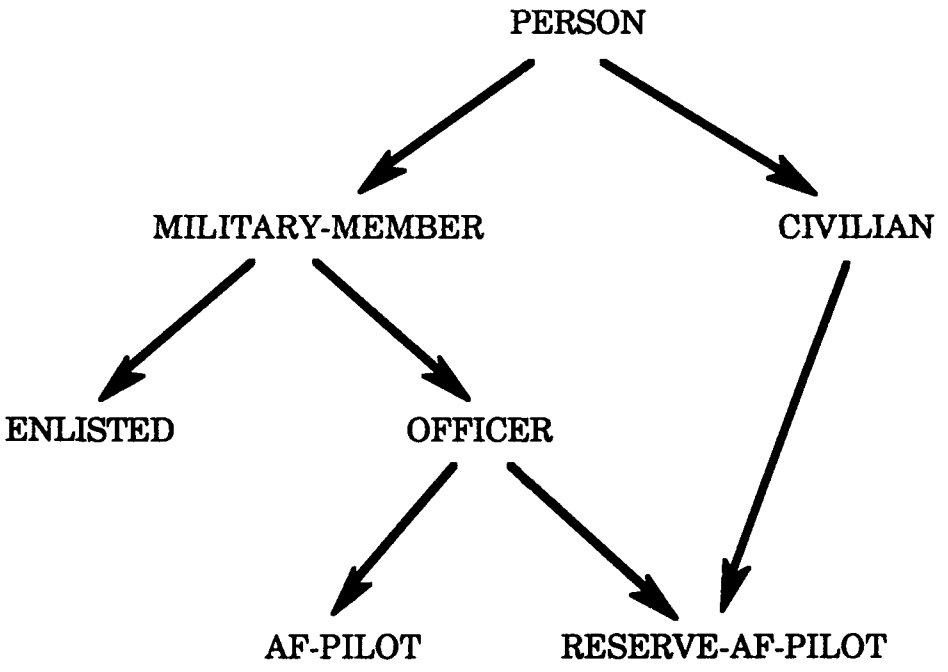


Figure 5. Inheritance Example.

A simple example of inheritance and multiple inheritance is depicted by Figure 5. Looking at the hierarchy we see that an AF-PILOT is an (ISA) OFFICER, an OFFICER is a MILITARY-MEMBER, and a

MILITARY-MEMBER is a PERSON. A RESERVE-AF-PILOT is a CIVILIAN as well as an OFFICER in the military, and inherits properties from both superclasses. All these classes in the hierarchy have similar variables associated with a PERSON (e.g. name, age, sex, etc.). All subclasses of MILITARY-MEMBERs have like variables, such as rank, time-in-service, etc., that differ from CIVILIAN instantiation variables. A service that may differ among all of these classes might be how that person's pay is calculated—the enlisted pay scale is different from an officer's; a pilot gets flight pay; and a reservist gets two paychecks. In this case the calculate-pay service would need to be defined at the lowest levels of the hierarchy: ENLISTED, AF-PILOT, and RESERVE-AF-PILOT. To avoid conflicting inheritance RESERVE-AF-PILOT would need to clarify which attributes it will chose to inherit from the multiply inherited attributes (like name and social-security-number) of both superclasses OFFICER and CIVILIAN. Another method would be to redefine or override the conflicting name and social-security-number attributes in the RESERVE-AF-PILOT class.

The physical organization of an object-oriented database also deviates from a relational database simply because of the many possibilities of storing specialized data types. Some of these objects may be large text data with a method that calls the editor or word processing application program in which it was created. This would allow modifications through that particular application and save the resulting document as a new data version. Other data that may require separate application programs to handle them are audio, video and graphical data objects. These present many possible applications where object-oriented techniques may shine.

1.5.2 Current Research Directions. This research essentially consists of an extended summary of current knowledge. Many areas of concern require continued research to better evaluate the benefits of this next generation object-oriented database system and its information modeling techniques. The following will show the major directions the current research has taken.

On the database evolution scale shown previously in Figure 1, we currently stand technologically between the second and third generation of databases. Current research appears to be progressing in two directions represented by the "Third-Generation Data Base System Manifesto" [48] and "The Object-Oriented Database System Manifesto" [1]. The former group seems to be progressing towards enhanced relational database research, by integrating object-oriented principles into current relational database technology. The latter approach (evolved from object-oriented programming languages) is to ignore the "limiting" relational database technology and develop totally new approaches to solving the object-oriented challenge. Cattell seems to think that both directions will continue, with the enhanced relational research aimed toward business applications, while the latter research targeted at the many scientific and specific object-based applications [10:235].

Many specific research areas have been isolated, with each expert seeing some as more important than the others. All, however, seem to agree that database system technology is not yet mature; aggressive research needs to continue at the fundamental level in order to solve the rapidly evolving database application problems of the all-too-near future.

"In addition to being important intellectual challenges in their own right, their solutions offer products and technology of great social and economic importance, including improved delivery of medical care, advanced design and manufacturing systems, enhanced tools for scientists, greater per capita productivity through increased personal access to information, and new military applications" [47:114].

Though there is a lack of standards in the field of object-oriented technology, much concern exists as to what and when standards need to be applied. Sometimes applying standards too early can stifle further progress in research. "It is a classical, and unfortunate, pattern of the computer field that an early product becomes the de facto standard and never disappears" [1:2]. One area, however, where standards would be beneficial is in object-oriented design and modeling. These areas provide more of a conceptual view of data modeling without affecting database research decisions.

The methodology of object-oriented database design is quite complex and remains an iterative practice due to the need for behavior modeling and class partitioning decisions. Many techniques will be covered in more detail throughout this thesis. As an example of the decision making activity, here is a general iterative process of just the object-oriented database schema design using six "easy" steps:

(Step 1) Initial Design: The user specifies a collection of classes and a set of constraints among them. Each class definition consists of a set of superclasses, a set of instance variables, and a set of methods.

(Step 2) Class Compilation: Each class definition is compiled so that it inherits instance variables and methods from its superclasses. During compilation, conflicts among inherited instance variables (and methods) and locally defined ones are resolved by a given set of conflict resolution rules. The compiled version of a class consists of only a set of instance variables and a set of methods, without a superclass declaration.

(Step 3) Schema Verification: The initial design is checked for consistency and non-redundancy. For each declared constraint, an appropriate verification should be performed. Typical verification tasks in this step are: "Is the set of constraints consistent?", When a user declares A [is-a] B, can A really become a subclass of B?", "Are there any equivalent classes or redundant IS-A relationships in the set of constraints?", etc.

(Step 4) Schema Interrogation: During the lifetime of a database, the user can query against a schema and the constraints on that schema. This step is necessitated by the need to understand non-obvious aspects of the current class hierarchy and to prepare for the next schema change operation. Typical queries raised by the user in step 4 are: "Is A a subclass of B?", What are subclasses of A?", "Is A disjoint with B?", etc.

(Step 5) Schema Modification: Typical schema change operations are: "create a new class", "create a new IS-A relationship among classes", "drop an existing class", etc. All the schema change operations in the taxonomy can be applied to the schema. Constraints on the schema also can be changed.

(Step 6) Schema Verification: The schema resulting from schema change operations and constraint modifications must be rechecked for consistency and non-redundancy. Typical verification tasks in step 6 are the same as those in step 3.

Steps 4, 5 and 6 are repeated iteratively during the lifetime of an object-oriented database. [33:32]

Many graphical methods of data modeling exist to assist in the design process. These methods are also automated in various computer-aided software engineering (CASE) tools. Modifications to old techniques seems to be the general trend. The latest of these techniques is the introduction of Objectcharts which are essentially extensions of Statecharts [15:9]. The variety of object-oriented data models being developed show the continuing effort to represent real world objects in a logical diagrammatic specification for object-oriented design.

1.6 Scope

This research will compare the current entity relationship modeling techniques with object-oriented modeling proposals to date. The underlying relational database management system (RDBMS) will also be compared to the proposed OODBMS. The military's key concerns of performance and reliability will be addressed with respect to the major areas of each system.

1.7 Approach / Methodology

Research will begin with an historical literature search of relational databases and entity-relationship modeling techniques. This search will try to uncover some of the major issues and concerns that were prevalent when the RDBMS construct was first proposed, and how they were resolved. A similar literature review will be accomplished on object-oriented modeling, current issues and future concerns of the various object-

oriented approaches proposed. An optimistic, though balanced, approach will be taken during comparison of both systems and evaluation of OODBMS expectations. The conclusion will entail recommendations to STARS regarding directions for future research and military implementations.

1.8 Document Summary

Chapter 2 describes the entity-relationship model and the relational model which semantically links it to the underlying relational database. This chapter also looks at the past and present concerns and accomplishments of the model and database to possibly predict about the future of the next-generation object-oriented field. Chapter 3 covers the current trend to expand the entity-relationship model and provide new modeling techniques for object-oriented database design. Several object-oriented modeling approaches will be reviewed along with their inherent trend toward unification of both software and database technologies and design paradigms. Analysis of object-oriented information modeling and the related database promises and concerns will be addressed in Chapter 4. Chapter 5 contains conclusions with respect to the thesis objectives and recommendations for further research.

II. Entity-Relationship and Relational Data Modeling

2.1 Overview

In this chapter we will cover both the entity-relationship and relational data models since they are key representations of the object-based and record-based logical models, respectively. Since its presentation by Chen [11] in 1976, the entity-relationship data model has become one of the most widely used and accepted object-based logical models in database design. "The [ER] model is as easy to use and understand as it is pictorial (i.e., graphical). It shows all types of higher-level abstractions, various relationships, and mapping constraints and cardinalities explicitly" [42:207]. The object-oriented data models, covered in Chapter 3, include many of the same concepts that are in the entity-relationship model. For example, both models are based on a view and collection of real world objects. The relational model uses a simple table format for its data structure and is mainly concerned with the actual values of the data to determine relationships. "The simplicity of the relational model is a benefit in ease of use and mathematical tractability, but is also a limitation" [10:5]. The limitations of the relational model, however, are overcome by the introduction of extended relational models such as the RM/T model (T for Tasmania, where it was first presented in 1979 [14:vi]) and the object-oriented models discussed in Chapter 3.

Before covering these data models we need to look quickly at the schema architecture of the database in order to understand the abstraction level(s) we wish to model. We also must understand the database life cycle to see where database modeling and design fit in.

2.2 Levels of Abstraction

From the history of database evolution (see Figure 1) we can see data independence as a common thread to the advantages realized as the semantic gap between application programmer and physical database (see Figure 2) is continually narrowed. This data independence covers both the physical and logical aspects of the database system. It is more easily understood when we look at the database from it's different levels of abstraction. Figure 6 distinguishes between the physically stored data structure, the logical view of that data, and the particular user views they want or are allowed to see. These database views relate to the internal, conceptual, and external schemas, respectively. The interfacing or mappings between these schemas is the job of the database management system and includes the user interface language, forms and/or menu. The following is a brief description of these three levels of semantic abstraction.

The internal level (sometimes called the physical level [35:4]) of abstraction contains data structure usage/definition information, their available access mechanisms, and record to address space distribution. The mapping from this schema to the actual database is accomplished by the DBMS using functions provided by the operating system.

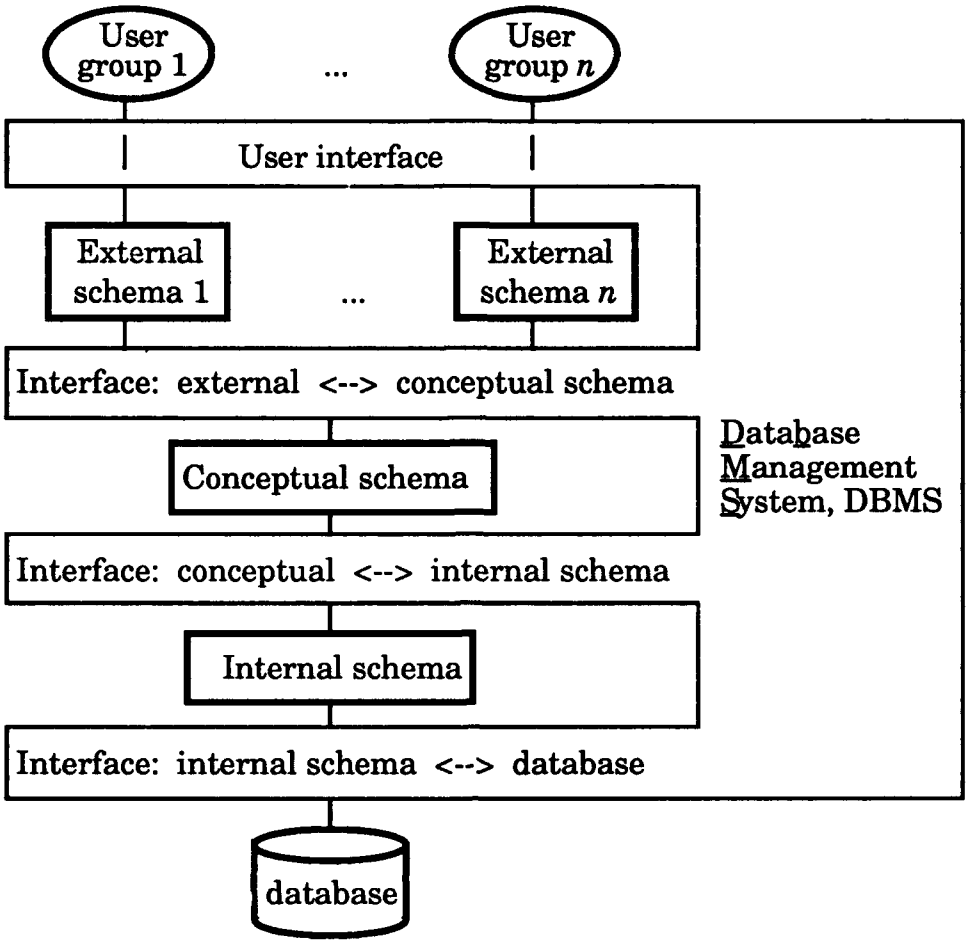


Figure 6. The Three-Level Architecture of a Database [50:12].

The conceptual level of abstraction is the main area where the data model is used to abstract the internal view by defining the information contents of the database in general, without reference to the data structure or access path definitions. The data model used has a "language" or conceptual tools to describe the database information in simple terms or structures. "The conceptual view is intended to serve as a neutral description from which the external and internal views are derived" [43]

The external level (or view level [35:4]) of abstraction provides individual views of the database using only those parts of the conceptual view that are of concern to or allowed to be seen by a particular user of the database.

This three-level architecture, presented by ANSI/SPARC in 1975 [35:20-21], provides three distinct advantages consistent with data independence. "First, the user or application system is freed from having to know where and how the data they use are physically stored. Second, heterogeneous hardware and software integration is simplified. Finally, the isolation of conceptual data definitions separate from their internal storage format allows for physical component replacement with minimal disruption" [43].

The DBMS handles all the interfaces or mappings between these schemas, as well as the interface to the user. Figure 7 shows the major components of a DBMS and how the formerly addressed three-level architecture of the database forms the three schemas incorporating the data dictionary. Table 1 indicates the association of these DBMS components to the actual interfaces shown previously in Figure 6.

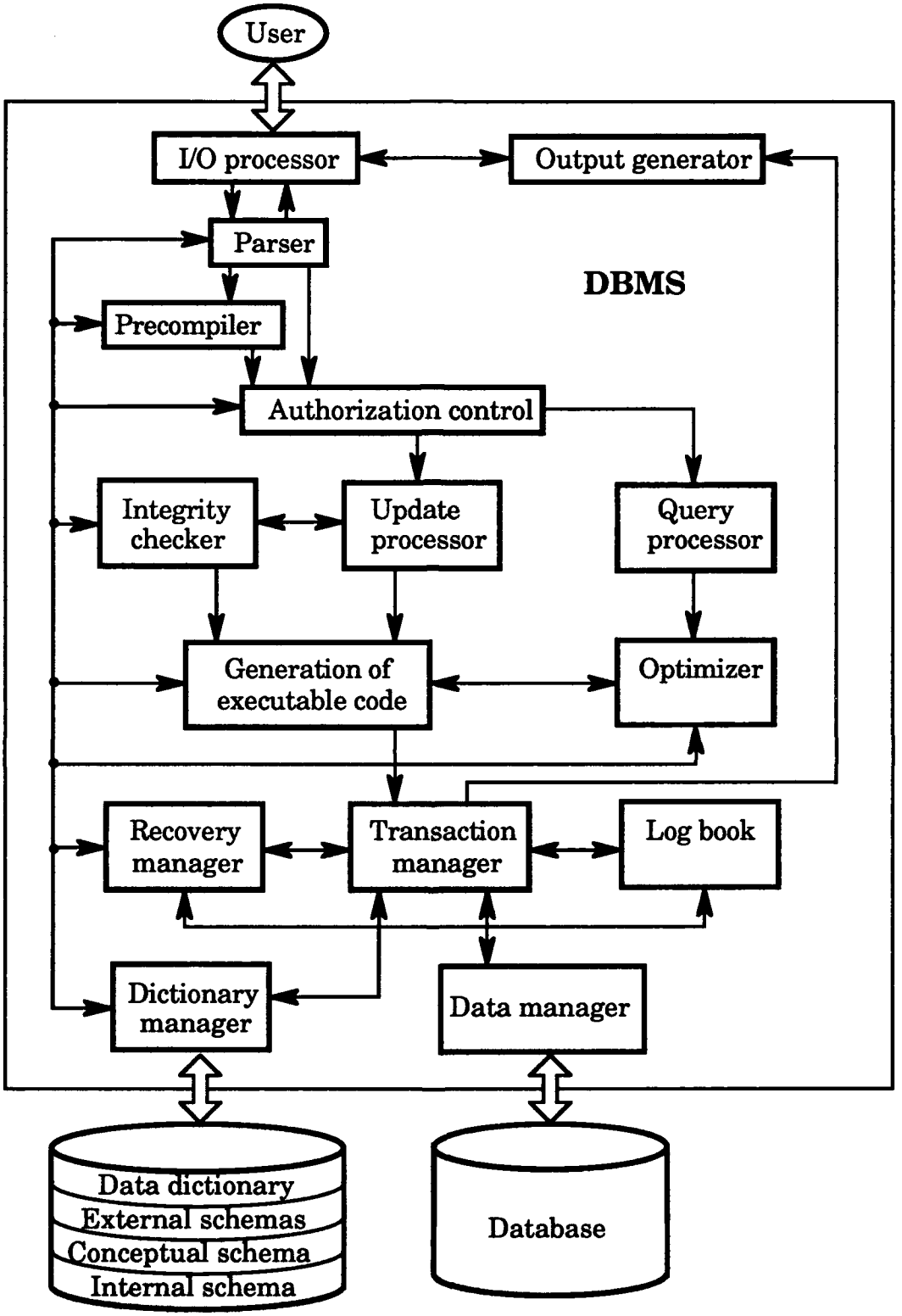


Figure 7. Organization/Components of a DBMS [50:24].

Table 1. Database Interfaces versus DBMS Components [50:27].

Interface	*Associated DBMS component(s)*
To the user	I/O processor (monitor)
External <--> conceptual level	Parser, precompiler, update and query processor
Conceptual <--> internal level	Code generator, optimizer
Internal level <--> database	Transaction manager, device and storage manager

2.3 Database Life Cycle

Entity-relationship data modeling and the relational model are described by Teorey [49] within the realm of the database life cycle. The following steps are his overview of the database life cycle and indicate when the entity-relationship and the relational model are used.

I. *Requirements analysis.* The database requirements are determined by interviewing both the producers and users of data and producing a formal requirements specification. That specification includes the data required for processing; the natural data relationships; and constraints with respect to performance, integrity, and security. It also defines the

hardware and software platform for the database implementation.

II. *Logical design.* The global schema, which shows all the data and their relationships, is developed using conceptual data modeling techniques such as ER. The data model constructs must ultimately be transformed into normalized (global) relations. The global schema development methodology is the same for either a distributed or centralized database.

a. *ER modeling.* The data requirements are analyzed and modeled by using an ER diagram that includes, for example, semantics for optional relationships, ternary relationships, and subtypes (categories). Processing requirements are typically specified using natural language expressions or SQL commands along with the frequency of occurrence.

b. *View integration.* Usually, when the design is large and more than one person is involved in requirements analysis, multiple views of data and relationships result. To eliminate redundancy and inconsistency from the model, these views must eventually be consolidated into a single global view. View integration requires the use of ER semantic tools such as identification of synonyms, aggregation, and generalization.

c. *Transformation of the ER model to SQL relations.* Based on a categorization of ER constructs and a set of mapping rules, each relationship and its associated entities are transformed into a set of candidate relations. Redundant relations are eliminated as part of this process.

d. *Normalization of candidate relations.* Functional dependencies (FDs) are derived from the ER diagram. They represent the dependencies among data elements that are keys of entities. Additional FDs and multivalued dependencies (MVDs), which represent the dependencies among key and nonkey attributes within entities, can be derived from the requirements specification. Candidate relations associated with all derived FDs and MVDs are then normalized to the highest degree desired by using standard normalization techniques. Finally, redundancies that occur in normalized candidate relations are analyzed further for possible elimination, with the constraint that data integrity must be preserved.

III. *Usage refinement.* In this step the goal schema is refined in limited ways to reflect processing requirements if there are obvious large gains in efficiency to be made. Usage refinement consists of selecting dominant processes of the basis of high frequency, high volume, or explicit priority: defining simple extensions to relations that will improve query performance; evaluating total cost of query, update, and storage; and considering the possible effects of denormalization. The justification for this approach is that, once local site physical design begins, the logical schema is considered to be fixed and is thus a constraint on efficiency. The database designer would like to remove this inflexibility if possible. Nevertheless, the usage refinement step makes assumptions about the physical design environment such that one may consider it to be actually an advanced stage of physical design.

IV. *Data distribution.* Data fragmentation and allocation are also forms of physical design because they must take into account the physical environment, that is, the network configuration. However, this step is separate from local schema and physical

design, because design decisions for data distribution are still made independently of the local DBMS.

A *fragmentation schema* describes the one-to-many mapping used to partition each global relation into fragments. Fragments are logical portions of global relations, which are physically located at one or several sites of the network. A data *allocation schema* designates where each copy of each fragment is to be stored. A one-to-one mapping in the allocation schema results in nonredundancy; a one-to-many mapping defines a replicated distributed database.

Three important objectives of database design in distributed systems are

- the separation of data fragmentation and allocation
- control of redundancy, and
- independence from local database management systems.

The distinction between designing the fragmentation and allocation schema is important: the first one is a logical mapping, but the second one is a physical mapping. In general, it is not possible to determine the optimal fragmentation and allocation by solving the two problems independently since they are interrelated. However, near-optimal solutions can be obtained with separable design steps, and this is the only practical solution available today.

V. *Local schema and physical design.* The last step in the design phase is to produce a DBMS-specific physical structure for each of the site databases and to define the *external user schema*. In a heterogeneous system, local schema and physical design are site dependent. The logical design methodology in step II simplifies the approach to designing large relational databases by reducing the number of data dependencies that need to be analyzed. This is accomplished by inserting ER modeling and integration steps (steps IIa and IIb) into the traditional relational design approach. The objective of these steps is an accurate representation of reality.

Data integrity is preserved through normalization of the candidate relations created when ER modeling is transformed into a relational model.

VI. *Database implementation, monitoring, and modification.* Once the design is completed, the database can be created through implementation of the formal schema by using the data definition language (DDL) of a DBMS. Then, as the database begins operation, monitoring will indicate whether performance requirements are being met. If they are not being satisfied, modifications should be made to improve performance. Other modifications may be necessary when requirements change or end-user expectations increase with good performance. Thus the life cycle continues, with monitoring, redesign, and modifications. [49:3-6]

2.4 Entity-Relationship Model

2.4.1 Basic Entity-Relationship Constructs. The three basic objects in the entity-relationship data model are entities, relationships, and attributes. An entity is an instance of a basic data object of interest. The term entities , or entity set, represents a collection of the same entity type. An interesting concept is that of the weak entity. The weak entities exist only because of their dependence on a dominant entity set.

Each entity is characterized by a set of attributes which either uniquely identifies the entity or describes specific characteristics about the entity. Due to its total dependence on the "parent" entity for existence, a weak entity can be modeled as a multivalued attribute within the dominant entity when there are no other relationships associated with the weak entity. For example, the PET entity of Figure 8 would be a weak entity

dependently related to the PET-OWNER entity, or it could be represented simply as attributes of PET-OWNER using the prefix P to distinguish the PET specific attributes of PET-OWNER. Sometimes it may be difficult to distinguish between an attribute or entity set. The distinction depends on the semantics of the attribute/entity set within the domain of the enterprise being modeled.

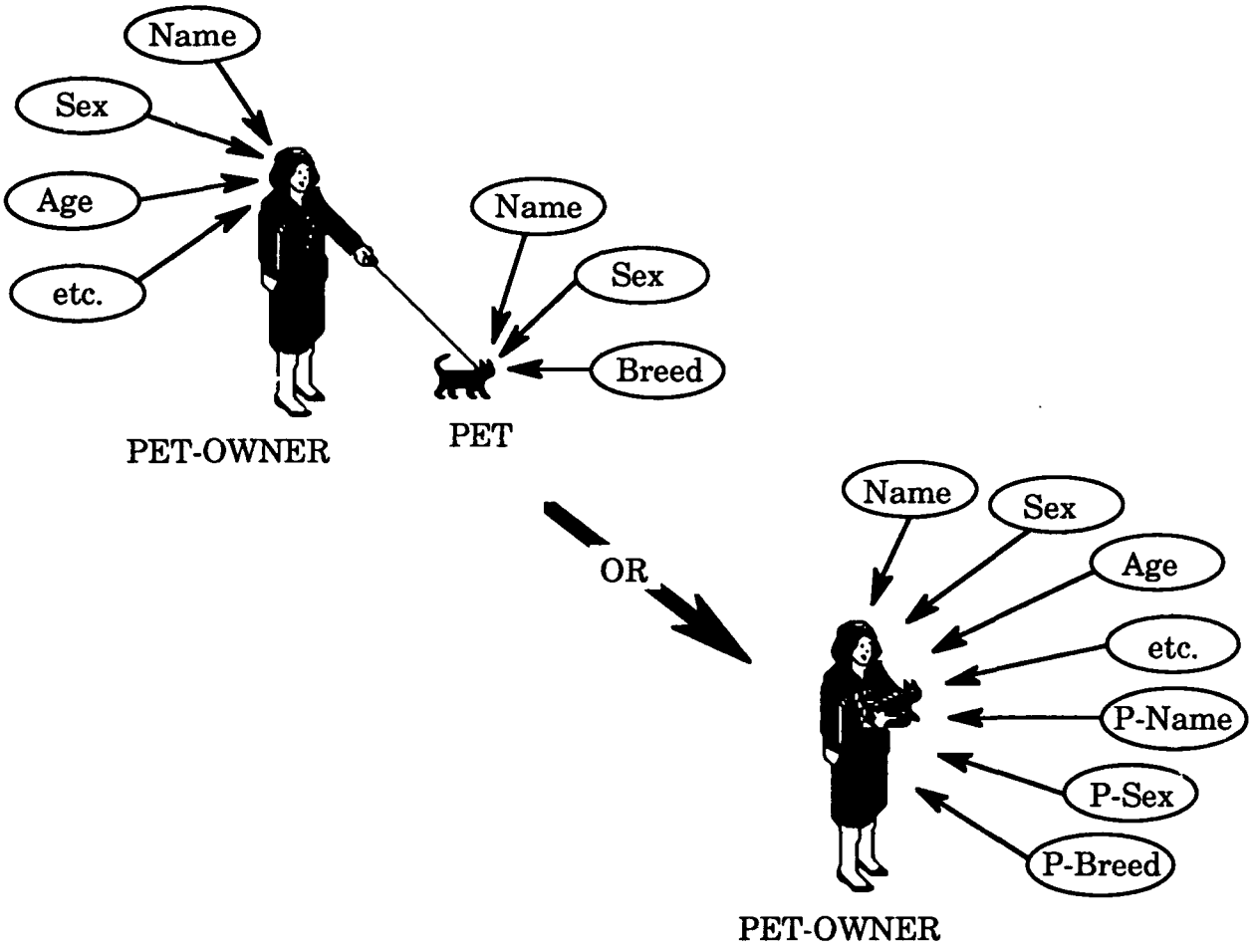


Figure 8. Weak Entity Example.

Finally, a relationship is an instance of a real-world association among several entities. Like entities, the term relationships, or relationship set, represents the collective relationship type. Relationships are distinguished by their degree, mapping cardinality, and existence.

The degree n of a relationship represents the number of entity sets associated with the relationship. It is rare to find n-ary relationships larger than three, so the first, second, and third degree relationships are referred to as unary, binary, and ternary, respectively. The unary relationship may seem unnecessary, but is used to represent recursive relationships among entity instances of a single entity set—for example, an employee manages another employee of the same entity set. The most common relationship is the binary relationship (association between two entities), while the ternary relationship relates three entities in such a way that it cannot be broken down into equivalent, and semantically pure, binary relationships.

Multiplicity of an association shows the number constraints on entities that can be in a relationship with a single entity. These mapping constraints for minimum and maximum cardinality fall within one of three basic constructs for the binary relationship: one-to-one, one-to-many, and many-to-many. The relationship's minimum cardinality is either zero or one, depending on its entities' optional or mandatory existence, respectively. The maximum cardinality of the relationship is either variable (by default) or an explicitly defined number.

The entity-relationship model has also been extended to represent the advanced database abstractions of the more sophisticated database

management system. The primary extensions that were made to the original entity-relationship model are grouped under generalization and aggregation. Generalization takes advantage of the similarities between entity types that, combined, make up a higher abstraction of the types. For example, both officer and enlisted personnel can be generalized into a common entity type called military-member. Aggregation makes it possible to show relationships to relationships. We can build higher levels of entity types consisting of a relationship between entities. An example of aggregation might be an entity called RESERVATION which consists of the relationship LISTED-ON between the entity PERSON and the entity FLIGHT.

2.4.2 Entity-Relationship Diagrams. A graphical representation of the entity-relationship model helps to visualize its overall logical structure. The graphical components of the entity relationship diagram are easy to learn and remain relatively consistent, though various authors tend to have a favorite approach to represent mapping cardinality. Figure 9 gives the complete entity-relationship diagramming notation as illustrated by Korth and Silberschatz [35]. The mapping cardinality technique is shown as taught by Roth [44].

The generic ER diagram shown in Figure 10 demonstrates the various constructs discussed earlier. Relationship set R1 shows the basic binary relationship between entity sets E1 and E2. The multiplicity on these relationships reveal that an entity in entity set E1 is associated with zero to five entities of entity set E2, while an entity in E2 is associated with at least one entity of E1.

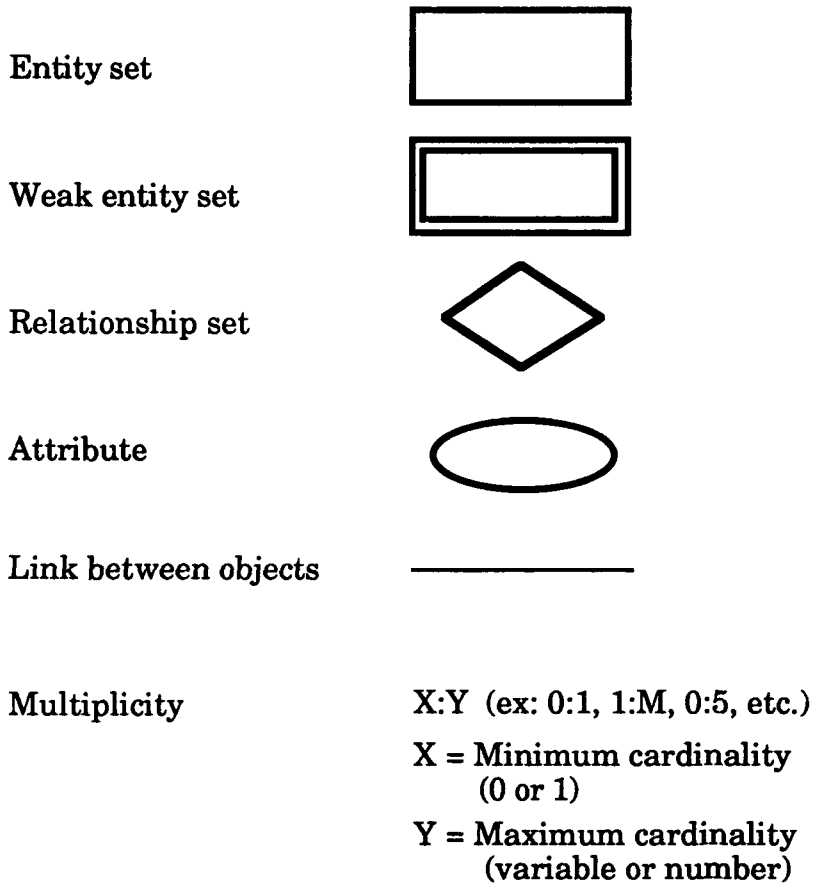


Figure 9. Entity-Relationship Diagram Notation.

An example of the unary relationship set, depicted by R2 and E2, would be parts that are made up of zero or more (0:M) other parts and, likewise, parts that make up one or more (1:N) other parts. The occasional ternary relationship is portrayed by the relationship set R4 among entity sets E5, E6, and E7.

E8 depicts a weak entity set, which is subordinate to a dominant entity from entity set E7; thus the relationship set R5 is shown here as a gray diamond to represent that it is an optional diamond notation.

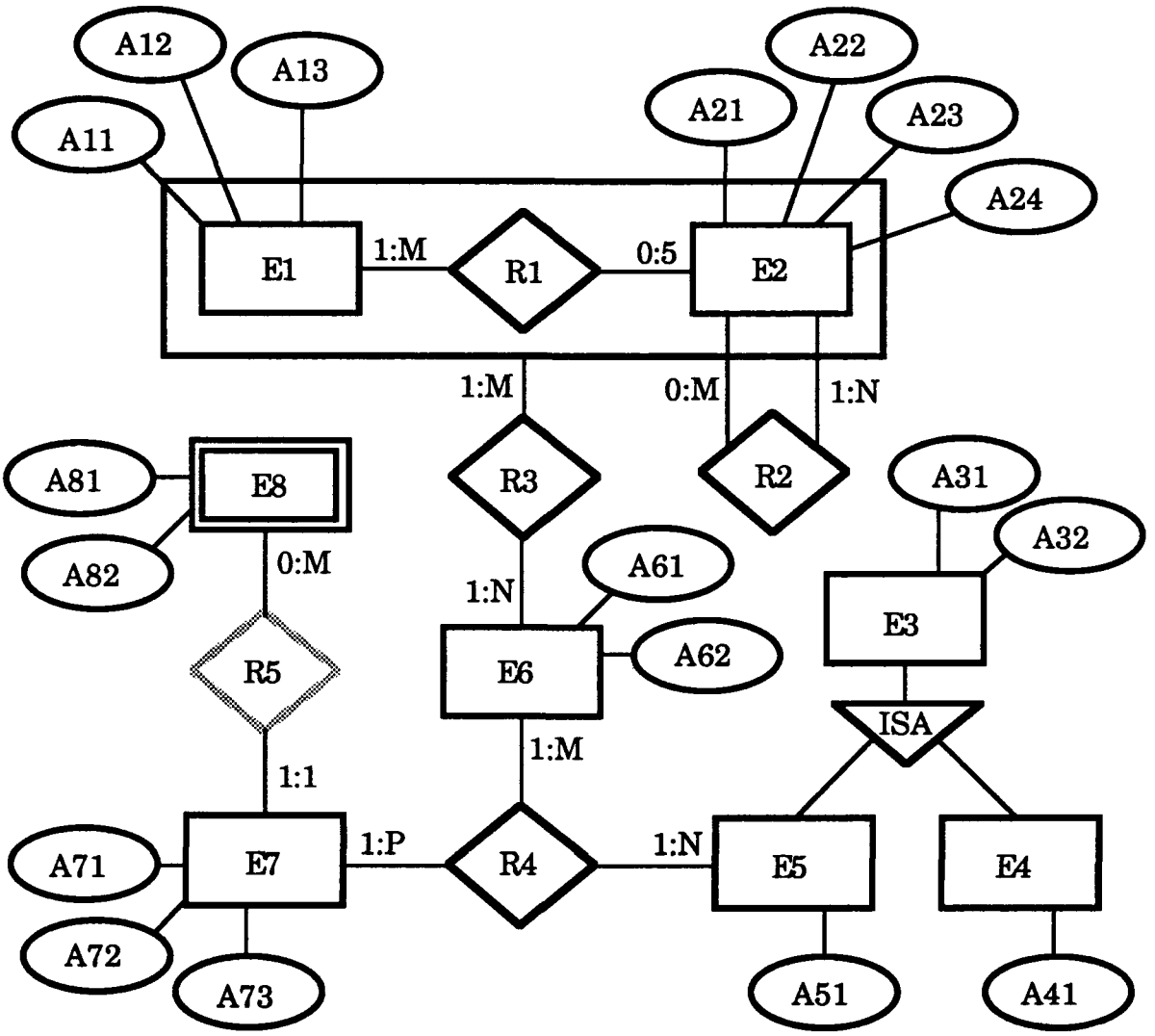


Figure 10. Entity-Relationship Diagram Example.

The relationship set R3 demonstrates the association of entity set E6 to the aggregate entity set consisting of the relationships R1 and their corresponding entity sets E1 and E2. Sometimes the line (link) between R3 and the aggregate box is shown extended to connect directly to the

relationship set R1. This researcher tends to think this action confuses aggregation with a ternary relationship construct.

The ISA relationship, depicted by the inverted triangle, exhibits the generalization of lower-level entity sets E4 and E5 into the entity set E3. Therefore, the attributes associated with E3 are directly inherited by both E4 and E5. If E3 had a link to some other entity, then E4 and E5 would also inherit that relationship.

The attribute bubbles may be left off the ER diagram to reduce clutter as long as they are accounted for using some other mechanism. An additional modification to the ER diagram, that has become popular from object-oriented systems analysis and design, is to eliminate the attribute bubbles and list them as a table within the enlarged entity rectangle. Figure 11 shows a simple example of this modification. Using this technique takes up less space and appears less cluttered when the diagram is of a much larger system.

2.5 Relational Model

Though it is a relatively new model, "the relational model has established itself as the primary data model for commercial data processing applications" [35:53]. The relational model is more semantically separated from the physical schema of the database, thus closing the semantic gap (see Figure 2) further, than the previous generation of network and hierarchical models. We'll take a quick look at the relational model because of its excellent representation of the underlying relational

database that is widely utilized today—at least in the business application domain.

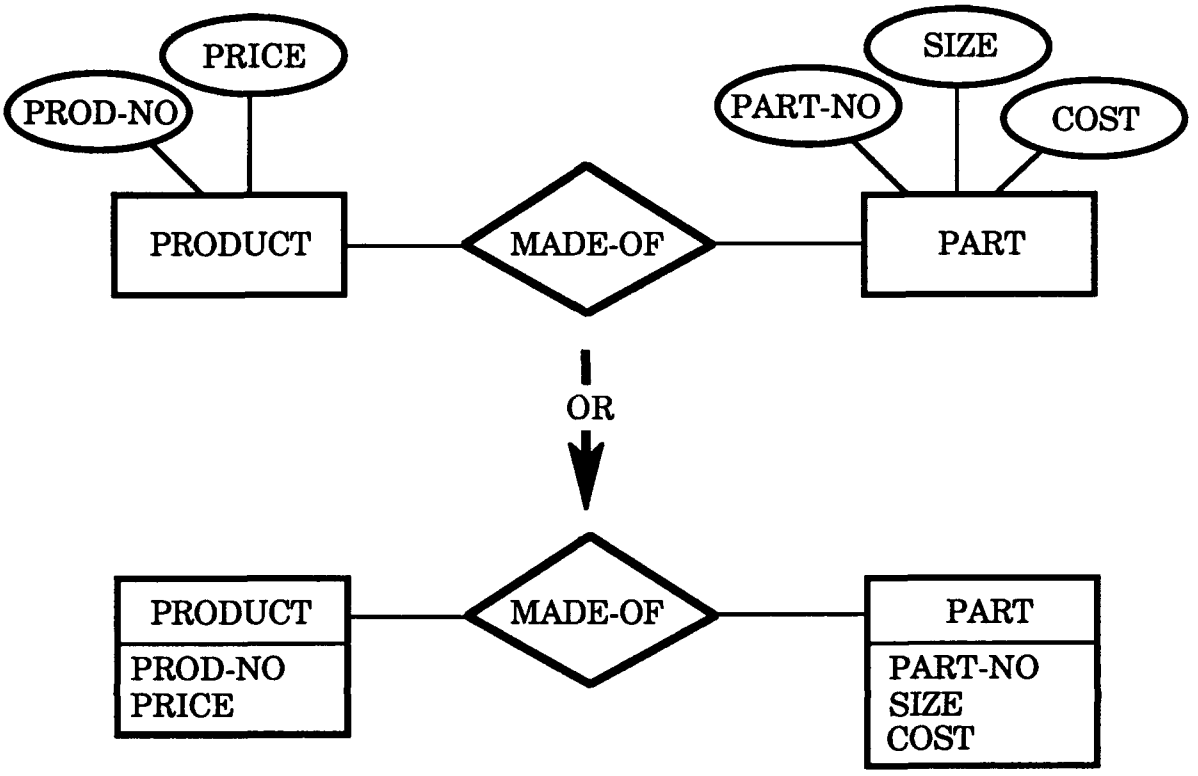


Figure 11. ER Diagram Attribute Modification.

The relational model takes its name from the semblance of the mathematical abstraction of relations and the image of tables from the relational database. This correspondence is natural when considering that each row (or tuple) in a table (or relation) represents an association among the values from each column (or attribute) in the row. Table 2 clarifies how the basic terminology is related between the ER model and the relational model and database.

Table 2. ER, Relational (Mathematical), and Database Terminology.

ER Model	*Relational Model*	*Relational Database*
Entity set, relationship set	Relation of degree n	Table with n columns
Entity, relationship	Tuple	Row of a table
Attribute	Attribute	Column of a table
Mapping cardinality	Cardinality of relation	Number of rows in table

E. F. Codd states that the link between the relational model and that of the mathematical relation set is based on two basic, yet essential, properties. "First, all of its elements are tuples, all of the same type. Second, it is an unordered set" [14:2]. Accordingly, he describes the relational model as dealing with the tuples by their information content, rather than by their ordering. Codd also stresses the discipline of not permitting duplicate tuples (or rows). This discipline maintains a more accurate database, considering the data is shared by many users. The mathematical foundation of the relational model allows the database such procedural and nonprocedural query languages as relational algebra and (tuple and domain) relational calculus, respectively. These, in turn, define the basic operations that make up the relational query languages found in today's commercial database systems.

Using the simple example found in Figure 11 we can create a typical relational model of the binary relationship set MADE-OF and the associated entity sets PRODUCT and PART. Figure 12 shows how the two relations (tables) are linked through an additional column of associated PROD-NO attributes in the PART table. Though rows may not be duplicated, a column can be repeated in another relation in order to form a connection/relationship between two tables. Note how it is mathematically possible to see that a product, such as number 950, is made up of parts numbered 123456 and 345678 simply by referring to PROD-NO column of the PART table. Though it may appear that part number 567890 is a duplicated row, the PROD-NO column has different values for each row, showing that this part is used in two different products. A collection of easy to understand tables, such as those shown here, is the essence of the relational model.

PRODUCT

PROD-NO	PRICE
447	349.95
853	27.50
567	299.99
397	786.25
950	99.99

PART

PART-NO	SIZE	COST	PROD-NO
123456	small	60.26	950
234567	large	9.25	853
345678	large	24.73	950
456789	medium	115.15	447
567890	large	50.00	447
567890	large	50.00	567
678901	small	71.00	447
789012	medium	17.85	853
890123	small	210.00	567
901234	large	705.00	397

Figure 12. Sample Relational Database.

2.6 *Early Concerns and Results*

"When the relational data model was first proposed, it was regarded as an elegant theoretical construct but implementable only as a toy. It was only with considerable research, much of it focused on basic principles of relational databases, that large-scale implementations were made possible" [47:113]. In two related papers, Date explodes thirty-four of the earlier "myths" of relational database management that he quickly clarifies as mostly misconceptions [17:77; 18:249]. Though some of the "myths" do "have a grain of truth in them, or are even entirely true, in the form in which they are stated" [17:79], they do show some of the early concerns with the new relational database technology.

Myth No. 1: A relational database is a database in which the data is physically stored as tables.

Myth No. 2: A relation is just a flat file.

Myth No. 3: The relational model is just flat files.

Myth No. 4: "Join" is just the relational version of the parent-child links found in hierarchic and network systems.

Myth No. 5: Relational systems must necessarily perform poorly.

Myth No. 6: The data must be hierarchically clustered for good performance.

Myth No. 7: Hierarchic clustering requires pointers.

Myth No. 8: The relational model presupposes (or requires) content addressable memory.

Myth No. 9: Relational databases use more storage than other kinds of database.

Myth No. 10: The relational approach is intended purely for query, not for production systems.

Myth No. 11: Automatic navigation does not apply to application programs, because such programs are forced to operate one record at a time.

Myth No. 12: Control is not possible in a relational system.

Myth No. 13: Relational systems provide no data integrity.

Myth No. 14: A relational system that included all necessary integrity controls would be indistinguishable from a hierarchic or network system.

Myth No. 15: Data has a naturally hierarchic structure.

Myth No. 16: Relational systems do not conform to the official database standard.

Myth No. 17: Hierarchic and network structures are more powerful than relational structures.

Myth No. 18: Relational databases involve a lot of redundancy.

Myth No. 19: The relational model is "just theory."

Myth No. 20: Relational databases require "third normal form."

Myth No. 21: The primary key concept is unnecessary.

Myth No. 22: The relational model is only suitable for simple data (i.e., numbers and character strings).

Myth No. 23: SQL (or QUEL or ...) is a panacea.

Myth No. 24: Database design is unnecessary in a relational system.

Myth No. 25: Third normal form is a panacea.

Myth No. 26: The relational approach is a panacea.

Myth No. 27: Updating must be done one record at a time.

Myth No. 28: Relational systems cannot handle the bill-of-materials problem.

Myth No. 29: Relational systems require the physical construction of numerous intermediate result tables.

Myth No. 30: Foreign keys undermine data independence.

Myth No. 31: Relational systems do not support the data dictionary concept.

Myth No. 32: Top-down ("entity/relationship") design and normalization are alternative and competing methodologies. Top-down design does not apply to relational databases.

Myth No. 33: The relational approach is just another technological fad. The relational model does not really differ in kind from the hierarchic and network "models."

Myth No. 34: "There's no such thing as a relational database."

Even during my last military assignment (as late as 1988), there was concern as our organization contemplated moving from the current networking database to a relational database system. The main issues of discussion were the relational database was easier to understand, yet perceived to be much slower than was achieved with the current network system.

Silberschatz et al. describe the results of the relational database concerns by showing how "the database research community extensively investigated the relational DBMS concept" [47:112-113]. They:

- Invented high-level relational query languages to ease the use of the DBMS by both the end users and application programmers. The theory of higher-level query languages has been developed to provide a firm basis for understanding and evaluating the expressive power of database language constructs.
- Developed the theory and algorithms necessary to optimize queries—that is, to translate queries in the high-level relational query languages into plans that are as efficient as what a skilled programmer would have written using one of the earlier DBMSs for accessing the data. This technology probably represents the most successful experiment in optimization of very high-level languages among all varieties of computer systems.
- Formulated a theory of normalization to help with database design by eliminating redundancy and certain logical anomalies from the data.
- Constructed algorithms to allocate tuples of relations to pages (blocks of records) in files on secondary storage, to minimize the average cost of accessing those tuples.
- Constructed buffer management algorithms to exploit knowledge of access patterns for moving pages back and forth between disk and a main memory buffer pool.
- Constructed indexing techniques to provide fast associative access to random single records and/or sets of records specified by values or value ranges for one or more attributes.

- Implemented prototype relational DBMSs that formed the nucleus for many of the present commercial relational DBMSs.

The results of this research are seen in the availability and popularity of commercial relational database systems today.

Two other areas where database research has played a vital role is in transaction management and distributed database systems. These areas became "hot issues" due to the popularity of the quickly spreading relational DBMS. With transaction management there were research breakthroughs in concurrency control and database system recovery. With the need of decentralized organizations came research developments in the distributed database arena to provide location transparency through distributed concurrency control algorithms. Crash recovery algorithms, query optimization technology, and multiple copy update algorithms were also designed for the distributed database environment. "Again we see the same pattern discussed earlier for relational databases and transactions, namely aggressive research support by government and industry, followed by rapid technology transfer from research labs to commercial products" [47:114].

2.7 Current Limitations and Prospects

The simplicity of the entity-relationship and corresponding relational data model gave rise to their wide acceptance and use, especially in the business arena. The current limitations of the relational database and related models lies in the ever growing need for more complex applications and the resultant complex data representation and storage. "Next-

generation database applications will have little in common with today's business data processing databases. They will involve much more data, require new capabilities including type extensions, multimedia support, complex objects, rule processing, and archival storage, and will necessitate rethinking the algorithms for almost all DBMS operations" [47:111]. Cattell echoes this sentiment when he states that "many applications' data are too complex to represent using relational tables and queries. The [next generation DBMSs] are designed to remedy this shortcoming, albeit at the expense of a more complex data model" [10:5].

2.8 Chapter Summary

We looked quickly at the schema architecture of the database in order to understand the abstraction level(s) we model, and glanced at the database life cycle to see where database modeling and design fit in. Once our foundations were set, we reviewed the entity-relationship and relational models to discover the simplicity of understanding they provide to the applications programmer, by shrinking the semantic gap between the real world view and the physical database. Note from Figure 2, that as the gap is closed, it moves from right to left; placing more and more responsibility on the shoulders of the database community.

As we have seen, "the [ER] model is frequently used for database design. However, it also serves well as a data model for any system that can be described as entities and relationships between them" [2:642]. The basic concepts of the ER data model have even extended into the realm of object-oriented data modeling, to include object-oriented systems analysis

and design as well as object-oriented modeling for database design. In Chapter 3, we will see modeling examples of the evolving object-oriented technology that promises to close the semantic gap even further. Even though the object-oriented field is a relatively new area where standards are absent, we will see that a single trend is developing.

III. Object-Oriented Data Modeling

3.1 Overview

From the evolution of database and application environments (see Figure 1) we have seen the transfer of more and more application concepts to the database management system. The obvious benefits from heading into the DBMS arena are the reuse of persistent objects and the data independence obtained therein. With this transfer of application concepts, however, there appears to be a merging of software application and DBMS design principles. Harel [27], Gupta and Horowitz [25], along with Dyer and Roth [20] see the steps for object-oriented software development applied just as effectively to a database management system. "Incorporating data-modeling techniques into the present framework [of systems development] could serve as an excellent melting pot for combining ideas from the world of data-intensive systems with ones from the world of reactive systems" [27:12]. "On the surface the object paradigm for software construction in general, and database management in particular, appears to be a marriage of software engineering principles that have been known for a long time" [25:3]. Though no "silver bullet" [9] exists for object-oriented database design, "the most comprehensive design methodologies applicable to object-oriented databases are those which were originally designed for general software development using object-oriented programming languages" [20:2]. Because of this natural merging of design methodologies, as both the database and application environments embrace the object-oriented

approach, it is difficult to address object-oriented data modeling aside from the design methods of object-oriented software development. Therefore, we will also address some of the object-oriented modeling and design methodologies for application systems that aid in database design.

We have chosen four representative object-oriented modeling and design methodologies to address specifically. The first methodology we will look at is the "state-of-the-art" [51:112] Object Modeling Technique proposed by Rumbaugh et al. [45]. This method provides a balanced approach to object-oriented modeling and design; it is expressive, yet clear and concise. The second analysis and design methodology covered will show the single diagram, multilayered approach of Coad and Yourdon [12; 13]. We will see that iterating through the layers of this single notation approach is itself the process of analysis and design. We will then look briefly at another well-known object-oriented design method advocated by Booch [7]. Here it will be clear that too much "breadth rather than depth" [51:110] of coverage, using several diagrams and notations, provides utility early in the design process, but restricts much needed detail later in the process. Finally, we will glimpse at one of the latest methodologies, proposed by Dyer [21], to bring back some of the required "depth" in object-oriented data modeling. Dyer's eclectic conceptual model for object-oriented database design builds upon the methodology of Hayes and Coleman in [29] while incorporating the constructs of Harel's Statecharts [26] and the Objectcharts of Coleman, Hayes and Bear [15].

Before addressing the object-oriented data models right away, we need to first look at the basic characteristics and requirements of object-

oriented data modeling and design. Given the history of adding database functionality to programming languages these criteria will naturally reflect many of the same aspects found in object-oriented application modeling and design.

3.2 Object-Oriented Data Modeling Characteristics

Just as the application and database design methodologies are converging, Cattell perceives the new data models—extended relational, functional, semantic, and object-oriented—as slowly combining their features into a single model [10:7]. This "single" model may not be one standard model, but may be based on one type of modeling that incorporates database design in the application design. Dyer and Roth contend that though there are a variety of object-oriented data models, the object-oriented software design methods are gaining acceptance as a comprehensive design technique, because object-oriented programming languages are realizing the need to support data modeling as well [20:2].

Since a real-world object—possibly having abstract internal structure and/or behavior—is difficult to model using a straight record-oriented data model; some form of object-oriented data model becomes crucial. Dennis Mcleod [25:16] describes four characteristics of an object-oriented database model and the database system that actualizes it.

1. **Individual object identity:** Abstract objects can be directly represented and manipulated in a database, independent of symbolic surrogates for them. Objects of various modalities (different media) can be accommodated.

2. **Explicit semantic primitives:** Primitives are provided to support object classification, structuring, semantic integrity constraints, and derived data. These primitive abstraction mechanisms support such features as aggregation, classification, instantiation, and inheritance. The roots of these semantic primitives are in the "semantic data models" and in artificial intelligence knowledge representation techniques.

3. **Active objects:** Database objects can be active as well as passive, in the sense that they can exhibit behavior. Various specific approaches to the modeling of object behavior can be adopted, such as an inter-object message passing paradigm, or abstract datatype encapsulation. The important point is that behavioral abstraction and encapsulation are supported, and procedures to manipulate data are represented in the database.

4. **Object uniformity:** All information (or nearly all) in a database is described using the same object model. Thus descriptive information about objects, referred to here as meta-data, is conceptually represented in the same way as specific "fact" objects.

These four characteristics are expanded upon by Booch [7] with a list of four major and three minor elements of the object-model. He asserts that the major elements are required to be considered object-oriented.

Major Elements:

- **Abstraction.** An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

- **Encapsulation.** Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics.

- **Modularity.** Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

- **Hierarchy.** Hierarchy is a ranking or ordering of abstractions.

Minor Elements:

- **Typing.** Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways.

- **Concurrency.** Concurrency is the property that distinguishes an active object from one that is not active.

- **Persistence.** Persistence is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created).

Each of these elements are refined, by Walker [51:103-106], into sub-topics which apply in varying degrees to particular systems or implementations.

Abstraction is further represented by two axes of stratification. The first divides the domain into layers of complexity and compositeness; and the second into layers of generalization (is-a-part-of) and specialization (is-a-kind-of).

With encapsulation we are concerned with the combination of an object's data (its state) and services (operations the object can provide). Associated with this is the object's interface with other objects or the messages to which it responds. These messages can be divided into three categories or protocols: public (response to external objects), protected (privileged access within a class or hierarchy), and private (internal to the object).

The decomposition of the application into sub-domains is introduced to avoid the "implementational flavor" associated with modularity. Here we recognize networks of closely coupled objects within the domain and either make them a super-object or identify them as a natural sub-domain of the application.

Along with single and multiple inheritance of class hierarchy come design questions with respect to reuse. Where is the appropriate place to insert a new class into the hierarchy? What is the strategy to derive abstract superclasses? The design method should help in the transition from design to implementation since object-oriented languages do not explicitly support hierarchies.

Typing has varying degrees and raises a problem with language independence and flexibility of design. Strong typing reduces flexibility and is usually implementation language dependent while weak typing is the opposite. The controversy continues; however, the implementation language should not depart drastically from the object-oriented paradigm when that design method is selected.

Problems of concurrency design are the same for object-oriented programming as for other languages. The autonomous behavior that exists among objects that can create their own threads of control must be modeled if the support is available.

Finally, persistence of an object must carry on to the class so that integrity is maintained. Since object identity implies persistence of relationships among objects, design must provide a means to avoid "dangling relationships" when an object is deleted.

Looking at other authors' perceptions of characteristics and requirements of the object-oriented paradigm, we see similar views. Rumbaugh et al. [45] list the major themes underlying object-oriented technology as abstraction (suspending commitments to detail); encapsulation (information hiding); combining data and behavior; sharing (inheritance); emphasis on object structure, not procedure structure; and synergy (among the concepts of identity, classification, polymorphism, and inheritance). Coad and Yourdon [12; 13] state the pertinent principles for object-oriented analysis and design as procedural and data abstraction; encapsulation; inheritance (portraying generalization-specialization); association (relationships); communication with messages; pervading methods of organization (object and attributes, whole and parts, classes and members identification); scale (to adapt to larger models); and behavior classification based on immediate causation, change over time, and similarity of functions. All these characteristics seem to expand upon Shlaer and Mellor's [46] three basic ideas described by an information

model: objects which pertain to the problem at hand, attributes of those objects, and relationships between the objects.

The overall purpose of modeling is to capture the specific aspects of a problem that are necessary to visualize the domain while reducing nonessential details that tend to limit our choices of implementation. The model must be clear enough to communicate your ideas to the customer, and descriptive enough for the designers to have a complete and consistent blueprint for construction.

3.3 Object Modeling Technique

The first approach we will look at is the Object Modeling Technique (OMT) of Rumbaugh et al. [45], because it builds upon the ER model (unifying application and database design) and is recognized by some as the most complete and consistent, state-of-the-art object-oriented design method [20:24; 28; 51:112].

While complaining about the invisibility and unvisualizability of software, Brooks states that "it has no ready geometric representation in the way that land has maps, silicon chips have diagrams, computers have connectivity schematics. As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs, superimposed one upon another" [9:1070]. Even in his article *Biting the Silver Bullet* [27]—triggered by Brooks' article *No Silver Bullet* [9]—Harel states that "regardless of how well devised it might be, one conceptual model might not be enough to take us from our initial thoughts to a final working implementation" [27:12]. These statements appear to be

true with respect to the OMT. In fact, the OMT methodology uses a total of three cross-linked graphical models to describe a system through all stages of development and implementation: the object model describes the abstract objects of the system (extended ER diagram); the dynamic model shows how those object interact (state diagram); and the functional model reflects how the system data values changes (data flow diagram). Each of these models will be addressed in more detail following a look at the OMT design methodology.

3.3.1 Object-Oriented Development. The overall OMT methodology consists of four stages of development where the three models seamlessly evolve from application domain modeling to system implementation modeling. The following is a list of the four stages along with the major steps and output document composition associated with each stage:

1. *Analysis*: Starting form a statement of the problem, the analyst builds a model of the real-world situation showing its important properties. The analyst must work with the requester to understand the problem because problem statements are rarely complete or correct. The analysis model is a concise, precise abstraction of *what* the desired system must do, not *how* it will be done. The objects in the model should be application-domain concepts and not computer implementation concepts such as data structures. A good model can be understood and criticized by application experts who are not programmers. The analysis model should not contain any implementation decisions. For example, a *Window* class in a workstation windowing system would be described in terms of the attributes and operations visible to the user.

Analysis steps:

1.1 Write or obtain an initial description of the problem (Problem Statement).

1.2 Build an Object Model.
Object Model = object model diagram + data dictionary.

1.3 Develop a Dynamic Model.
Dynamic Model = state diagrams + global event flow diagram.

1.4 Construct a Functional Model.
Functional Model = data flow diagrams + constraints.

1.5 Verify, iterate, and refine the three models.

Analysis Document = Problem Statement + Object Model + Dynamic Model + Functional Model.

2. *System design*: The system designer makes high-level decisions about the overall architecture. During system design, the target system is organized into subsystems based on both the analysis structure and the proposed architecture. The system designer must decide what performance characteristics to optimize, choose a strategy of attacking the problem, and make tentative resource allocations. For example, the system designer might decide that changes to the workstation screen must be fast and smooth even when windows are moved or erased, and choose an appropriate communications protocol and memory buffering strategy.

System design steps:

2.1 Organize the system into subsystems.

2.2 Identify concurrency inherent in the problem.

2.3 Allocate subsystems to processors and tasks.

2.4 Choose the basic strategy for implementing data stores in terms of data structures, files, and databases.

2.5 Identify global resources and determine mechanisms for controlling access to them.

2.6 Choose an approach to implementing software control.
2.7 Consider boundary conditions.
2.8 Establish trade-off priorities.

System Design Document = structure of basic architecture for the system as well as high level strategy decisions.

3. *Object design*: The object designer builds a design model based on the analysis model but containing implementation details. The designer adds details to the design model in accordance with the strategy established during system design. The focus of object design is the data structures and algorithms needed to implement each class. The object classes from analysis are still meaningful, but they are augmented with computer-domain data structures and algorithms chosen to optimize important performance measures. Both the application-domain objects and the computer-domain objects are described using the same object-oriented concepts and notation, although they exist on different conceptual planes. For example, the *Window* class operations are now specified in terms of the underlying hardware and operating system.

Object design steps:

3.1 Obtain operations for the object model from the other models.
3.2 Design algorithms to implement operations.
3.3 Optimize access paths to data.
3.4 Implement software control by fleshing out the approach chosen during system design.
3.5 Adjust class structure to increase inheritance.
3.6 Design implementation of associations.
3.7 Determine the exact representation of object attributes.
3.8 Package classes and associations into modules.

Object Design Document = Detailed Object Model + Detailed Dynamic Model + Detailed Functional Model.

4. *Implementation*: The object classes and relationships developed during object design are finally translated into a particular programming language, database, or hardware implementation. Programming should be a relatively minor and mechanical part of the development cycle, because all of the hard decisions should be made during design. The target language influences design decisions to some extent, but the design should not depend on the fine details of a programming language. During implementation, it is important to follow good software engineering practice so that traceability to the design is straightforward and so that the implemented system remains flexible and extensible. For example, the *Window* class would be coded in a programming language, using calls to the underlying graphics system on the workstation. [45:5, 261-263]

3.3.2 Object Model. The object model describes the objects of the system. Each of the three models may vary in importance depending on the application domain. However, the object model is the most crucial of them all, since we are emphasizing the object-oriented paradigm. As an extension of the ER data model, the object model represents objects, the relationship between them, their attributes, and the services they provide. The graphical notation of the object model—as in the dynamic and functional models—provides an easy-to-understand formality that is coherent, precise, and concise. As we discuss the object diagram you can reference the complete notation shown in Figures 13 and 14.

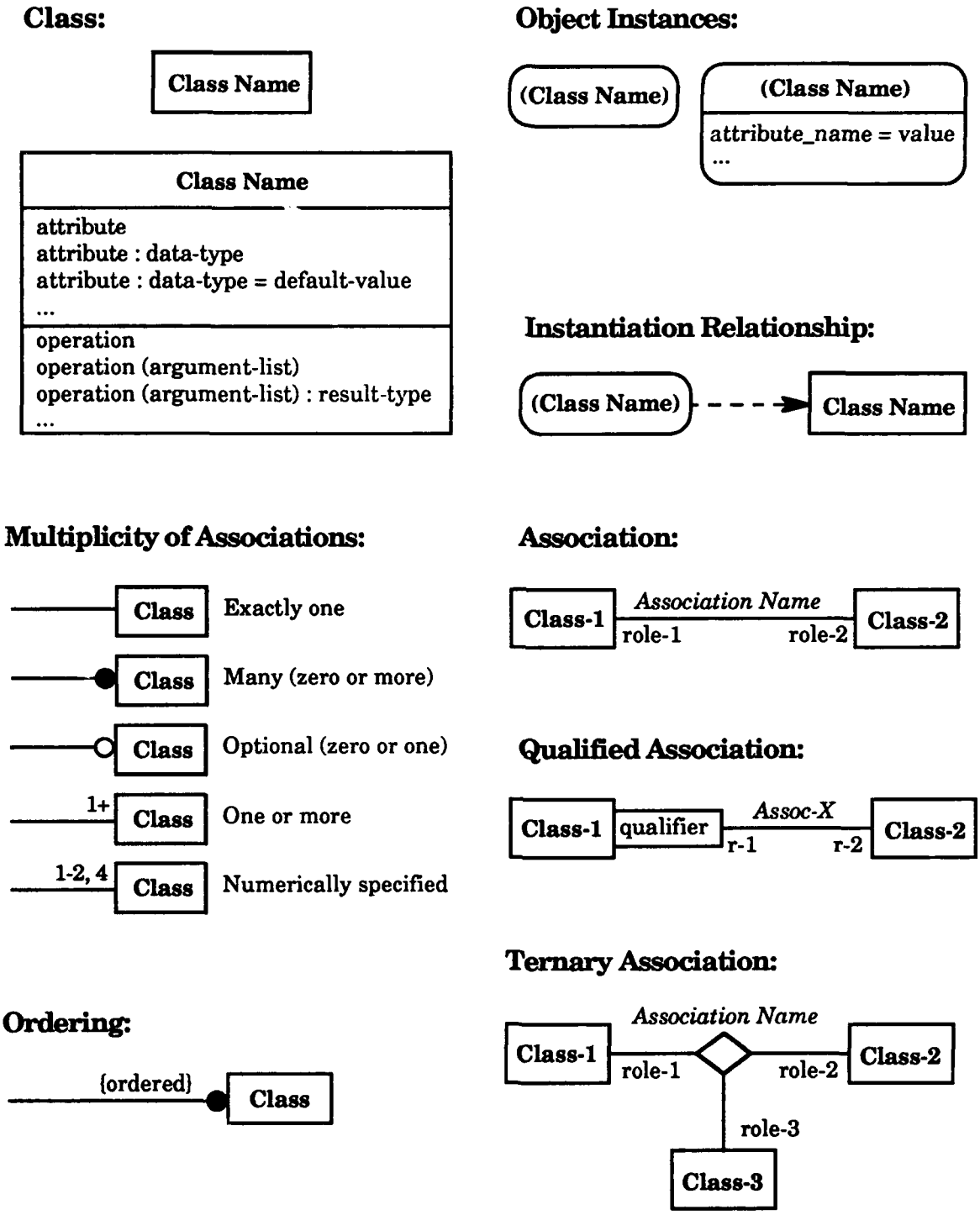


Figure 13. Basic Object Model Notation [45:inside cover].

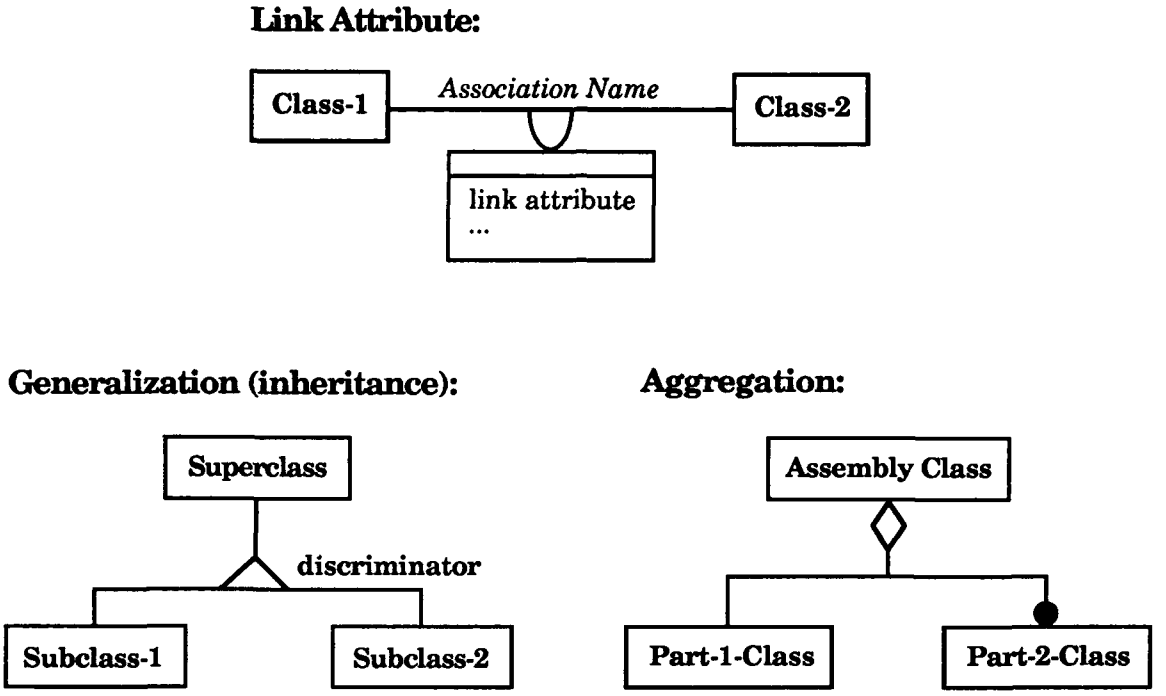


Figure 14. Extended Object Model Notation [45:inside cover].

The notation for object instances allows for two types of object diagrams: the class diagram and the instance diagram. The instance diagram would be helpful when there is interest in a specific scenario; however, the typical object diagram would use the class constructs.

As Figure 13 suggests, attributes and services (operations) of the object class may or may not be listed under the name; this depends on the abstraction level of the diagram. When they are listed, it should be in the region order shown: class name, attributes, and operations. Note the optional type and default details shown for attributes and the signature (argument list and result type) details for services. It is important to be

consistent with the service signatures when a behavior has services on several classes.

Just as in ER diagramming, OMT relationships are shown as labeled lines between classes (associations) or instances (links). The multiplicity symbols located at the line ends show cardinality of the association. The symbology directly relates to similar notations, such as Shlaer and Mellor's [46] arrowheads, or the X:Y notation of the ER model described in Chapter 2. Though these associations may seem pertinent only to database modeling, Rumbaugh et al. asserts that they "are a useful modeling construct for programs as well" [45:31].

Figures 13 and 14 also indicate several advanced relationship concepts that are summarized in Table 3. Though not directly shown by Figure 14, modeling an association as a class is depicted by expanding the link attribute box into the recognized three-region box of an object class.

Generalization and inheritance refer to the relationship among classes in an 'ISA' class hierarchy and the method of sharing attributes and operations of the superclasses in the hierarchy. Generalization/inheritance is a fundamental concept of the object-oriented paradigm, is a popular design abstraction mechanism and promotes code reuse during implementation. A subclass may override a service of the superclass (or superclasses, if multiple inheritance), but should never override the signature of a service. For example, the rotate service for a circle class, inherited from a figure superclass, would be overridden with a null operation.

Table 3. Advanced Link and Association Concepts.

Concept	*Description*
Link attribute	A property of the links in an association
Association as a class	When links can participate in associations with other objects or when links are subject to operations.
Role name	Indicates a direction across an association
Ordering	Explicitly shows an ordered set of objects exists on the "many" side of an association
Qualification	Reduces the effective multiplicity of an association to disjoint subsets; a form of ternary relationship
Aggregation	A tightly coupled form of association. Each assembly-part association is an aggregation

Constraints on the objects, classes, attributes, links, and associations of an object model are declared within braces near the affected entity. The "{ordered}" notation shown in Figure 13 is a good example of a constraint notation. Constraints are expressed using equations or natural language (e.g. "{salary ≤ $100,000}" to cap employees wages). Rumbaugh et al.

contends that a "good" object model should also contain many constraints without compromising simplicity [45:74].

Based on the object model constructs briefly discussed here, it is easy to see that this model would have high significance in the object-oriented database design arena as well as in the application domain. Refer to [45] for further study of these and other advanced topics concerning the object model and OMT in general.

3.3.3 Dynamic Model. Once the static structure of a system is understood through the object model, the dynamic model shows us how the objects change state over time. A state diagram for each "dynamically important" class is necessary to capture the timing and control issues of the system. The combined state diagrams—modeling the events and states of each class—make up the dynamic model. The dynamic model relies heavily on the "visual formalism" of Harel's [26] state diagram and (when nesting is required) Statechart notation. "To be useful, a state/event approach must be modular, hierarchical and well-structured" [26:233]. OMT's dynamic model provides such a generalization hierarchy for states and events to share structure and behavior. Figure 15 shows the complete dynamic model notation, and can be referenced as each construct is addressed.

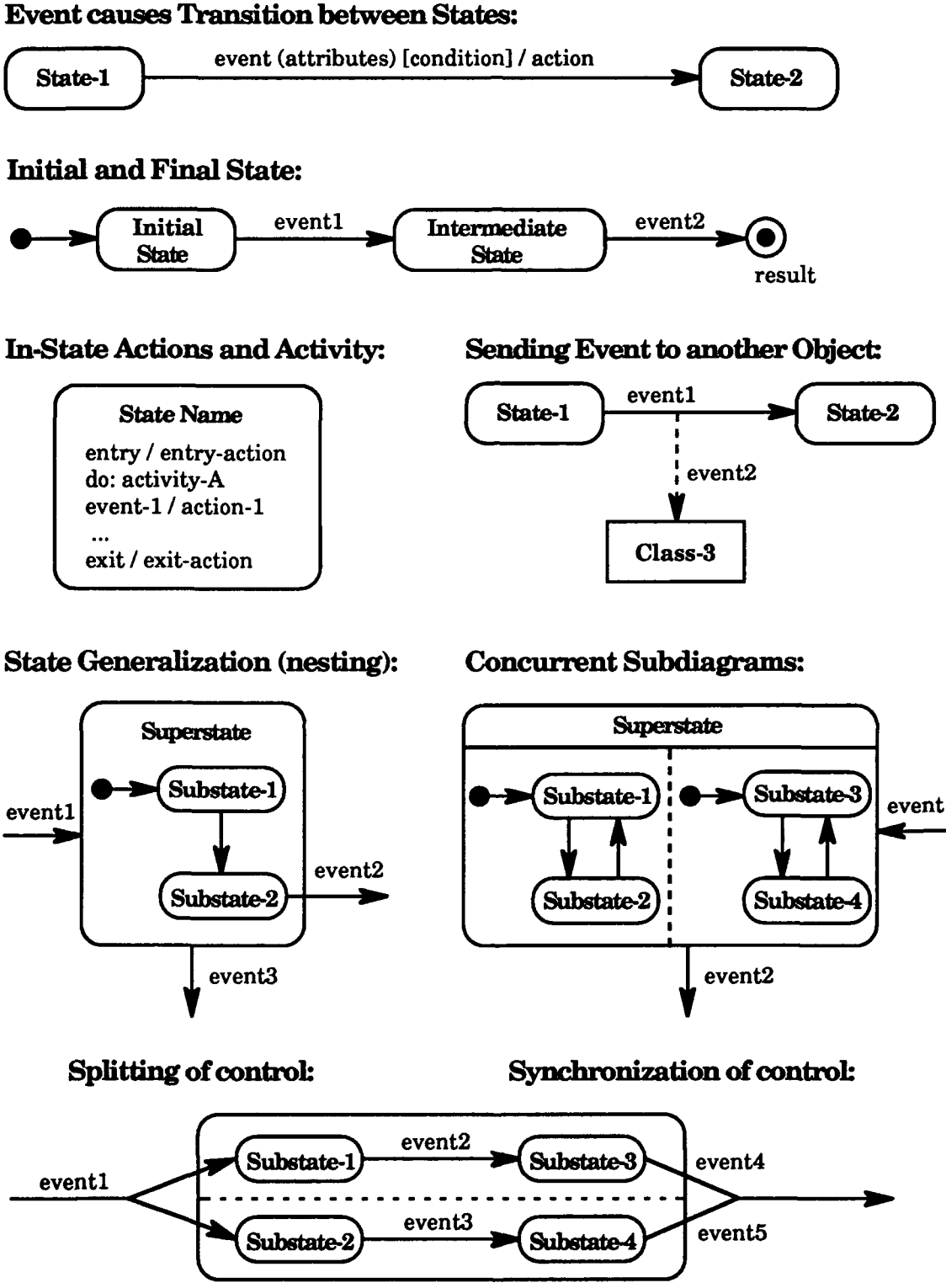


Figure 15. Dynamic Model Notation [45:inside cover].

As seen from the notation, events can have attributes (data values conveyed by the event), conditions (guard that must be true for transition), and cause an action on transition between states. Though not explicitly shown in Figure 15, an action on transition can also be another event. A corollary to this is that an object/class' action on transition can be sending an event to another object/class.

Within the state, activities can occur and are listed after the 'do' reserved word. As shown in the notation, entry and exit actions can also be used to show all the transitions entering and exiting the state. Entry/exit actions listed within the state can reduce diagram clutter from showing the same actions on multiple incoming (or outgoing) transitions. As a bonus, "they make self-contained state diagrams possible for use in multiple contexts" [45:113].

The dynamic model provides for nesting state diagrams to allow for a natural decomposition to lower-level state diagrams. Aggregation of concurrent state diagrams is also possible through similar use of Harel's [26] rounded box notation to cluster subdiagrams, and a dotted line between them to show the partitions. Synchronization of concurrent activities is also modeled in a similar fashion by splitting control when necessary, and later merging that control.

3.3.4 Functional Model. After representing the who and what of the system through the object model, and the when and where of system state changes through the dynamic model, we can use the functional model to represent the how of system data transformations. Computations within

the system are expressed through an extended data flow diagram (DFD) notation that Walker suggests is the "least successful" aspect of OMT [51:111]. Hayes and Coleman state that "DFDs are an inappropriate choice for the functional model since the flow of data through processes partially determines an order of computation" [29:175]. In order to better show system behavior, rather than a particular computation, they prescribe specifying formal pre-post conditions to supplant DFDs [29:176]. Robert France, shows that semantically extended DFDs can be used to formally reason about behavioral properties while benefiting from the flexibility and intuitive appeal of the popular DFD [22:329]. Whichever way the functional model ends up being tailored or replaced, Walker agrees "that OMT seems to make as much sense of DFDs in the object-oriented context as any previous attempt" [51:112].

The complete notation for the data flow diagrams making up the functional model can be viewed in Figure 16. The typical DFD specifies the meaning of operations and constraints through the use of processes, data flows, actor objects (terminators), and data store objects. Unlike the object model and the dynamic model, a functional model does not organize data values into objects or show control information (though control information can be repeated on the DFD). As a collection of DFDs, the functional model is useful for nesting functionality of a system by showing its breakdown into smaller functional units. Constraints can also be used in the functional model to specify restrictions on operations.

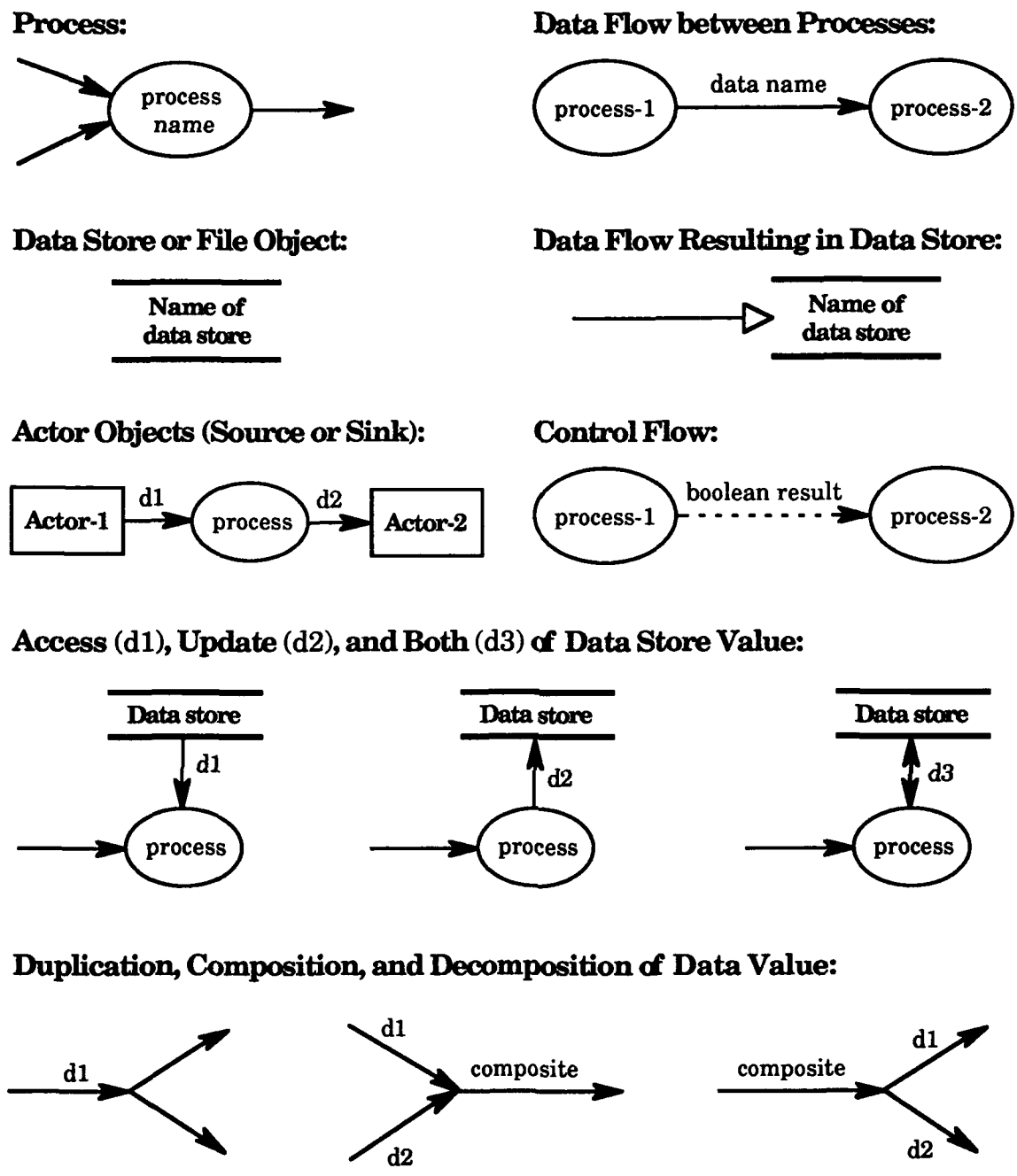


Figure 16. Functional Model Notation [45:inside cover].

A process, at its lowest level, is a simple operation/function on a single object without the occurrence of side effects. Side effects can occur,

however, at higher-level processes if they contain actor or data store objects (such as the DFD itself). The input and output data arrows can be labeled according to their role or type of value. The actual implementation of processes is through the services (methods) of an object class.

Data flows connect objects or processes, simultaneously showing one's output value and another's input value. As Figure 16 suggests, data flows can clone into duplicate values, divide an aggregate value, or combine into one. While data flows show intermediate values of a computation, they do not change them. Data flows shown as dotted lines are used to represent the optional control flows of a DFD. The Boolean values shown on control flows are not input values themselves, but simply control process execution. Actors (otherwise known as externals) are sources or sinks of initial input and final output data flows of the functional model

The data store shows data values at rest, or that there is a delay between the generation and use of data. Like actors, the data stores are treated as objects. The hollow arrowhead notation provides a new construct to show a data store object creation. The data flow value using this notation is then treated as an object.

OMT's combination of the three models (object, dynamic, and functional) provide an overall picture of the system from different views. Depending on the application being designed, one model may have more importance than another. From the database modeling standpoint we would mainly be concerned with the object model view. In fact, Dyer and Roth say the simple identification of persistence to the type information

would complete OMT as a design methodology for both object-oriented languages and object-oriented database systems [20:23-24].

3.4 Object-Oriented Analysis and Design

Coad and Yourdon's overall five layer (analysis), four component (design) object-oriented model [12; 13] is graphically depicted by Figure 17. The five layers represent five activities (not necessarily sequential steps) of object-oriented analysis (OOA) that show a gradual move from higher levels of abstraction to more detailed levels. The five layers are iteratively cycled through as desired until analysis and specification are complete. The four components of object-oriented design (OOD) are likewise four activities or vertical slices of the model, and not sequential steps. They are used to improve and add to the five layered OOA results while maintaining the stability of the problem domain's organizational framework over time. As we address the portions of Coad and Yourdon's single diagram, multilayered model you can reference the appropriate construct in the summarized notation of Figure 18.

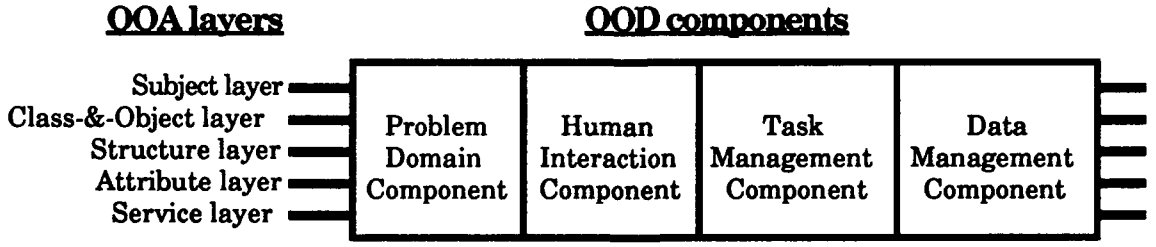


Figure 17. Five Layer, Four Component Model [13:26].

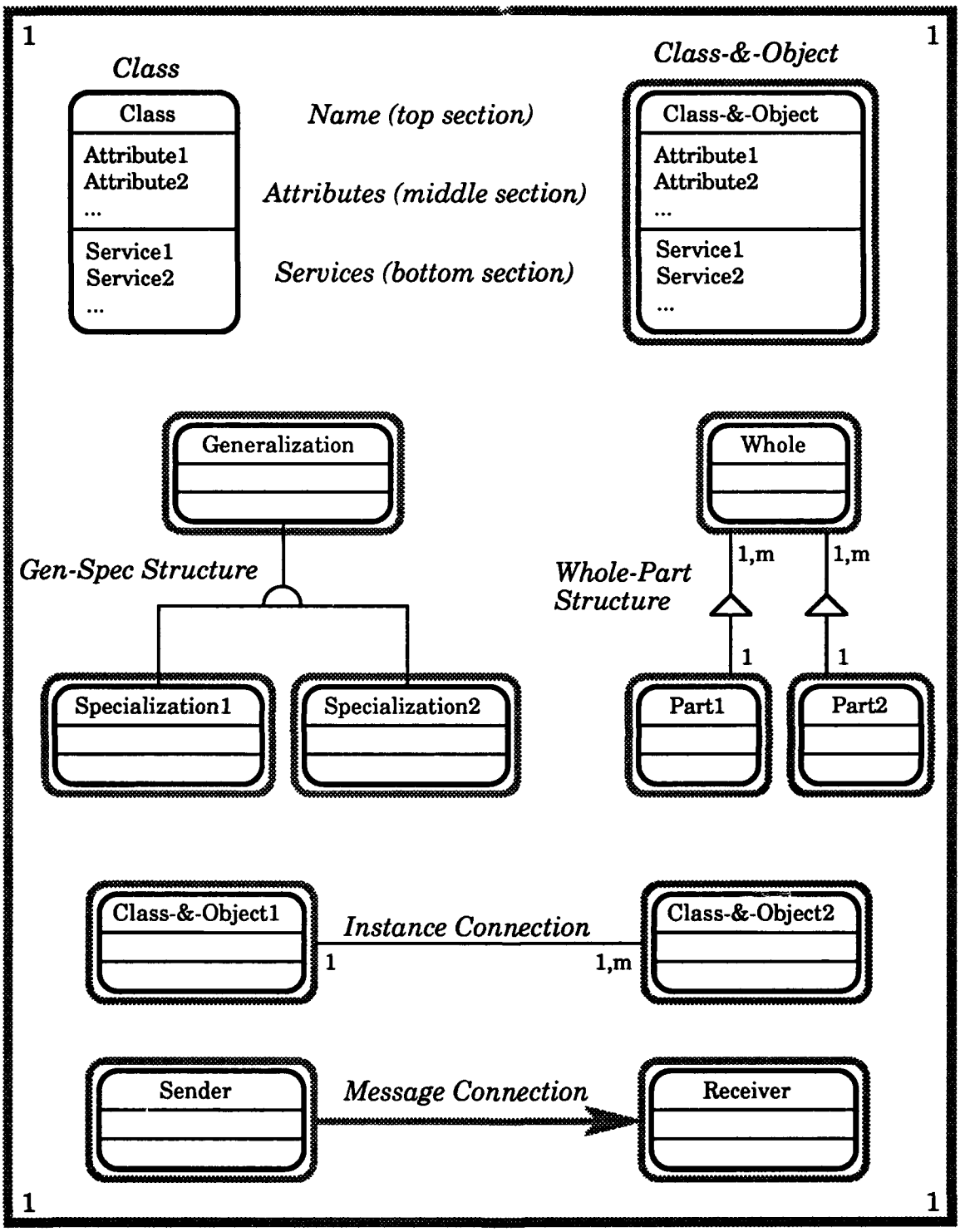


Figure 18. OOA/OOD Notation Summary [13:35].

3.4.1 Five Layers. The "initial" layer is the Subject layer which identifies the major Subjects (subsystems) of the problem domain. To prevent information overload inherent in large complex models the Subject layer shows parts of the overall system in more digestible bites. The notation is simply a rectangular box, with the Subject name and a number inside (see Figure 19), for the "collapsed" Subjects. An example of the "expanded" notation is shown as the outer box in Figure 18, with the Subject number placed in all four corners. This is used when shown with the other layers of the model. A partially expanded Subject notation exists that simply lists the Class-&-Objects within a Subject under the Subject name and number of the collapsed Subjects.

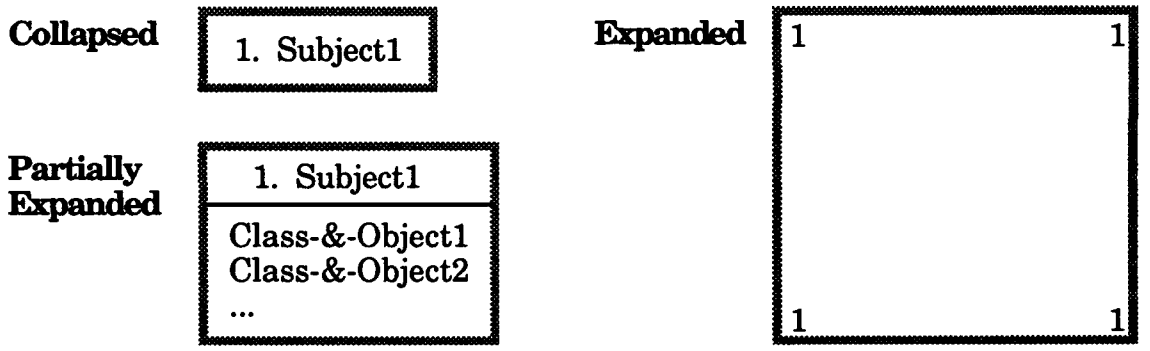


Figure 19. Subject Notation [12:112].

The Class-&-Object layer is where we identify the classes (and their objects) of the problem domain in order to better technically represent the conceptual view of the real world in our system. "Another motivation for emphasizing Class-&-Objects is our desire to create a stable framework for analysis and specification" [12:54]. Though their attributes and services

may change over time, rarely do the classes and objects of a system. There is also the simple Class notation to represent a generalization class whose complete set of objects are portrayed by its specializations using the Object-&-Class notation. An example of this would be a Generalization-Specialization (Gen-Spec) structure, where all the Objects of a Generalization Class are reflected only in the Specification Class-&-Objects because each specification has one or more differing attributes or services. If the "StandardThing" specification Class-&-Object had no added attributes or services beyond the "Thing" generalization Class, as shown in Figure 20, it could be absorbed into the "Thing" generalization Class, changing it to a Class-&-Object construct.

In the Structure layer we focus on the complexity of, and the relationships between Class-&-Objects in the system. The two structures of concern are the Gen-Spec ("is-a-kind-of") and the Whole-Part ("is-a-part-of") structures. Note that the notation lines of the Gen-Spec structure relate classes while the Whole-Part structure lines relate objects. The number or range marked on the Whole-Part structure line shows multiplicity of each object that are allowed to make up, or be made up of, another object. The markings are also read backwards from the typical ER mapping cardinality. For example, the Whole-Part Structure notation of Figure 18 indicates that "Whole" objects can be made up of one or more (1,m) "Part" objects, while each "Part" object is a part of only one (1) "Whole" object.

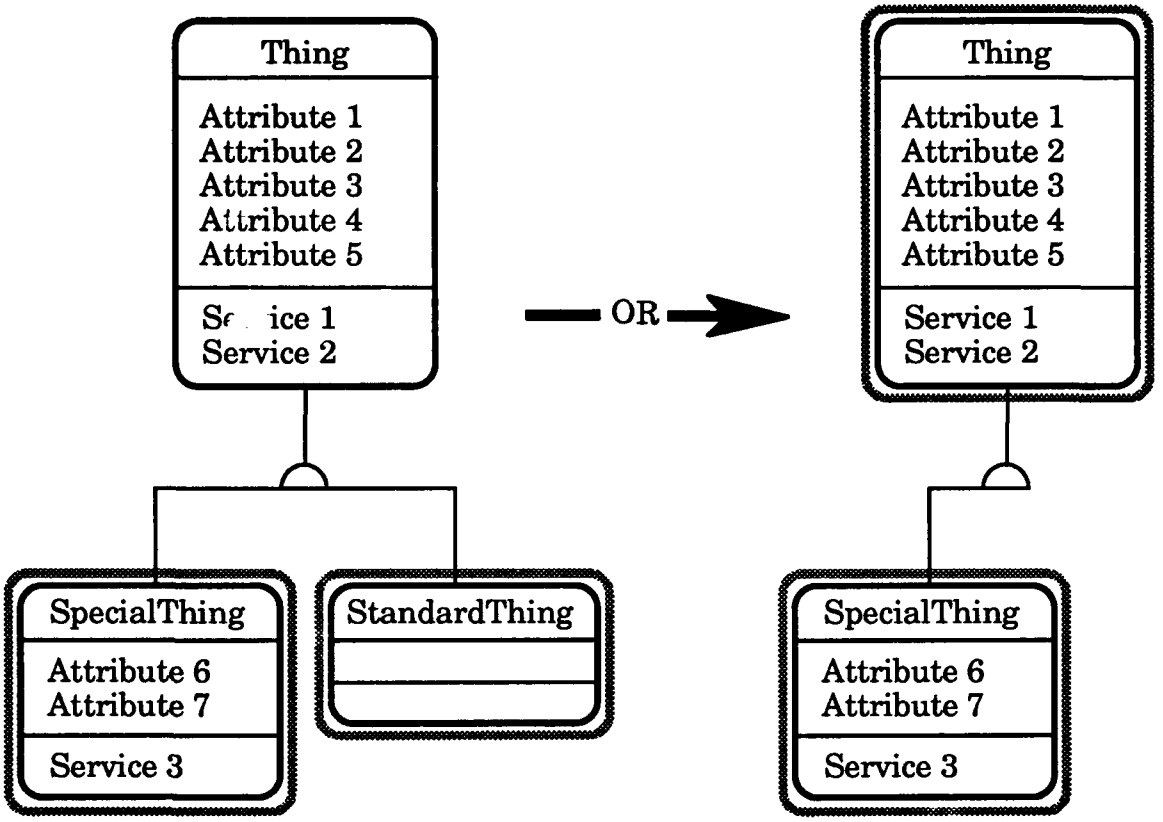


Figure 20. Using Class or Class-&-Object Generalization.

The Attribute layer adds more detail to the Class-&-Object layer by identifying class characteristics. The notation is to simply list the Attributes in the center section of the Class and Class-&-Object symbols. "Attributes describe values (state) kept within an Object, to be exclusively manipulated by Services of that Object. We treat Attributes and exclusive Services on those Attributes as an intrinsic whole" [12:120]. Attributes are not available for direct manipulation from outside sources, except through the Services provided by the Class-&-Object. Along with listing Attributes, the Attribute layer identifies Instance Connections to show associations between Objects (not Classes). Though like the ER relationship notation,

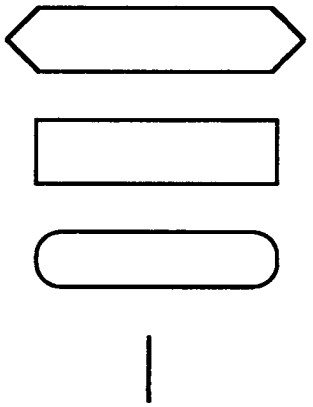
the Instance Connection line is not labeled. The mapping cardinality is also placed on the line endpoints as an amount or range, and again reads differently than the ER notation, as demonstrated in the Whole-Part structure discussion.

Class-&-Object behavior is identified by the Service layer, where Services are listed in the lower third area of the Class and Class-&-Object box. As stated previously, Services are the only "methods" available to the outside world, or to other objects, whereby the state of an Object can be altered or queried. In order to better figu e out what services must be specified the object states need to be identified. This can be done through an Object State Diagram that shows the Attribute values that change with respect to behavior change. Coad and Yourdon use the Object State Diagram notation shown in Figure 21 to show only states and legal transitions.

Figure 21. Object State Diagram Notation [12:146].

Once the required services are identified, and listed in the appropriate Class-&-Object box, the message connections must be noted (as in Figure 18) to show the process dependencies (need for Services) of Objects. Coad and Yourdon also use a Service Chart notation (see Figure 22) to more formally specify Services. Though this notation is much like flow charting, it is considered "ideal...for applying the principle of

procedural abstraction systematically, within the limited context (scope) of a single Service" [12:158].



Condition (if; precondition; trigger, terminate)

Text block

Loop (while; do; repeat; trigger/terminate)

Connector (connected to the top of the next symbol)

Figure 22. Service Chart Notation [12:157].

For documentation purposes the Service Chart would then be placed appropriately within a Class-&-Object template as outlined by Figure 23. Though the exact standard is not important, establishing a standard documentation template such as this is important [23:32].

3.4.2 Four Components. The notation of Figure 18 remains the same for the four components of the OOA/OOD model. In fact, the "first" component of OOD starts with the results of OOA. The Problem Domain Component (PDC) is key to keeping the stability of the problem domain base throughout analysis, design, and implementation. Any modification to the PDC is a tradeoff between expressing the modification criteria and maintaining the stable representation of the problem domain. Modifications consequently affect the potential reusability of the model with

other systems of similar problem domain. As you cycle through the four components of OOD the PDC will most likely change due to changing requirements, technology (i.e., inheritance capability of the implementation language), early lack of understanding during OOA, etc.

specification
attribute1
attribute2
...
external input
external output
object state diagram
additional constraints
notes
service1 <name & **service chart**>
service2 <name & service chart>
...
and as needed,
traceability codes
applicable state codes
time requirements
memory requirements

Figure 23. Class-&-Object Template [12:156-157].

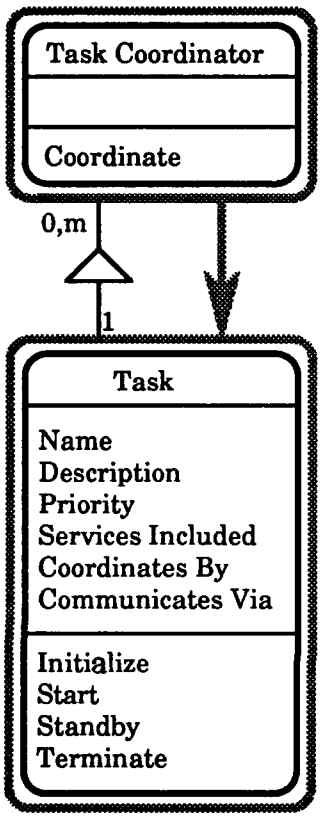
The Human Interaction Component (HIC) looks at the people who use the system and the user interface that is required for them to take advantage of the maximum system capabilities. This component hits the "touchy feely" area of software engineering because "Design decisions affect people. An individual's emotions and mental perceptions may be positively or negatively affected. And organizational behavior (i.e., corporate culture) may change, too" [13:57]. Once the customers are classified (by skill level, organizational level, and/or group membership) it is good to develop task

scenarios from their point of view and figure a general command hierarchy. Prototyping may be a large factor in the HIC due to the I'll-know-it-when-I-see-it mentality of many people. The emotional satisfaction of the client with the user interface is the primary concern of HIC.

The Task Management Component (TMC) is where decisions to task or not to task are made. While simplifying the design and implementation of needed concurrent behavior, tasking adds complexity to design, coding, testing, and maintenance when not necessary. Tasking decisions involve identifying event-driven tasks, clock-driven tasks, and establishing priority and critical tasks. It is important to keep tasks to a minimum and coordinate them. The notation used for task objects is the Whole-Part structure with Message Connections. See Figure 24 for the TMC notation and task definition templates.

The Data Management Component (DMC) is where the database management system infrastructure is provided, isolating the impact of the actual database management scheme. Coad and Yourdon recommend including the design of the data layout and corresponding Services into the DMC, according to the data management system implemented [13:84]. As we have seen through our discussion, object-oriented design methods, such as this and the Object Modeling Technique of Rumbaugh et al. [45] provide the opportunity to extract traditional relational or network data structures from the object-oriented design. The visible extension of the ER model gives us this capability. As Dyer and Roth suggest for the OMT [20:23], simply identifying persistence in the DMC, by indicating it on the diagram, is all

that is needed to model designs implemented in object-oriented languages or database systems.



Notation Template

Task Definition Template

Name
Description
Priority
Services included
Coordinates by
Communicates via

Figure 24. TMC Notation and Task Definition Templates [13:77].

3.5 Booch's Object-Oriented Design

Booch has several models of object-oriented design that are expressed using a variety of diagram notations and documentation templates. The different models of Booch's object-oriented design are summarized in Figure 25. Though the notation is designed to cover a wide range of system types and all problem domains, not every aspect of OOD must be used at all times. Walker feels that "Booch's diagram notation has sacrificed detail or

depth for breadth" and is limited "in exposing the detailed semantics of the application" [51:110]. However, he does feel that Booch's object-oriented design makes "very substantial contributions towards meeting the needs of the designer" [51:112].

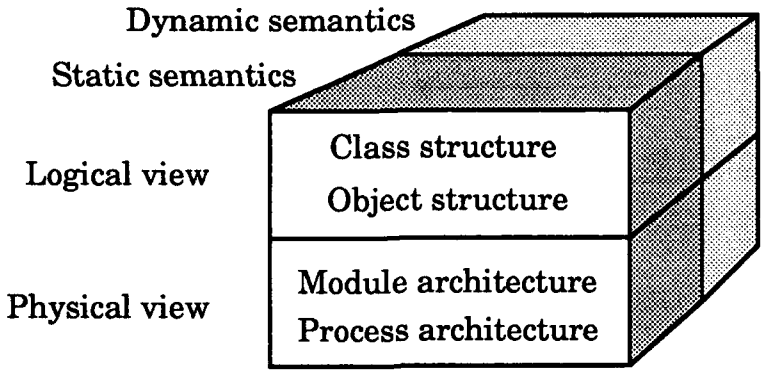


Figure 25. Models of Object-Oriented Design [7:155].

As Figure 25 suggests, the logical view of a system contains the class and object structures, and are depicted by related class and object diagrams. The physical model is interested in the module and process architecture (and associated diagrams), that portray the implementation software and hardware components. Moreover, these four diagrams represent the static model of a system while the dynamic semantics of design are expressed through additional state transition and timing diagrams. Each of these diagram constructs and notations will be covered in turn.

The overall process of object-oriented design proposed by Booch is described as "round-trip gestalt design" [7:188]. This process style is

described by incrementing and iterating through refinement of the various system views, and is based on the following four events:

- Identify the classes and objects at a given level of abstraction.
- Identify the semantics of these classes and objects.
- Identify the relationships among these classes and objects.
- Implement these classes and objects.

This process is completed when "there are no new key abstractions or mechanisms, or when the classes and objects we have already discovered may be implemented by composing them from existing reusable software components" [7:190].

3.5.1 Class Diagrams. The class diagram shows what classes exist and how they relate to each other. The appropriate notations (called icons) are shown in Figure 26, and should be referenced as needed during the discussion.

The amorphous blob or cloud icon represents a class abstraction. It has a dotted/dashed outline to show that operations are generally done on the object instances, rather than the class itself. The relationship and cardinality notation seems extensive and more complicated than necessary. However the identification of an "undefined" relationship is handy for early class conceptualization. The class utility icon represents one or more free subprograms, which are "procedures or functions that serve as nonprimitive operations upon an object or objects of the same or different classes" [7:82].

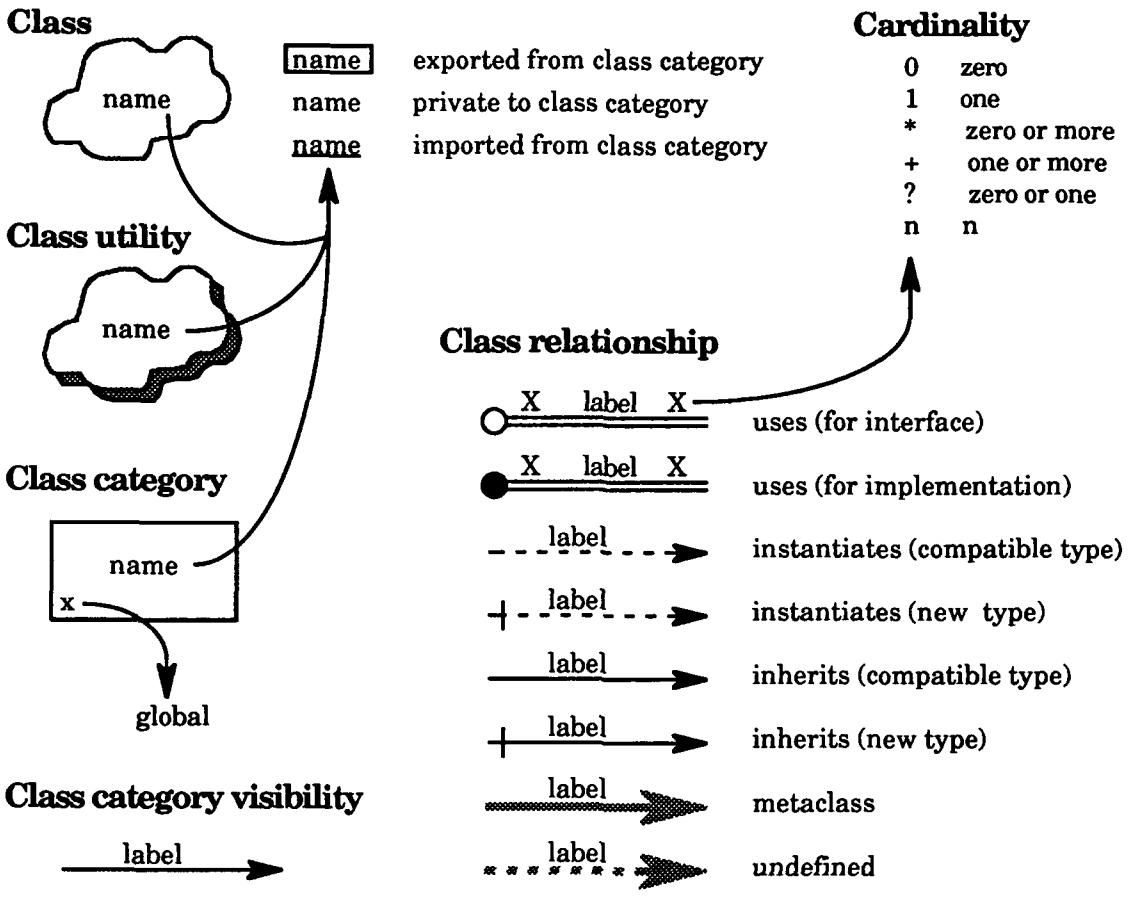


Figure 26. Class Diagram Icons [7:inside cover].

When grouping classes into subject areas to reduce diagram clutter the class categories notation can be used. The relationship arrow shows dependencies on other class categories by pointing to the class category from which classes are imported. Instead of multiple arrows, the word "global" would simply be placed in the lower left corner of the class category box to show that all its exported entities are imported by all the other class categories. Finally, the class diagram templates shown in Figure 27 are used to provide documented substance to the graphic notation.

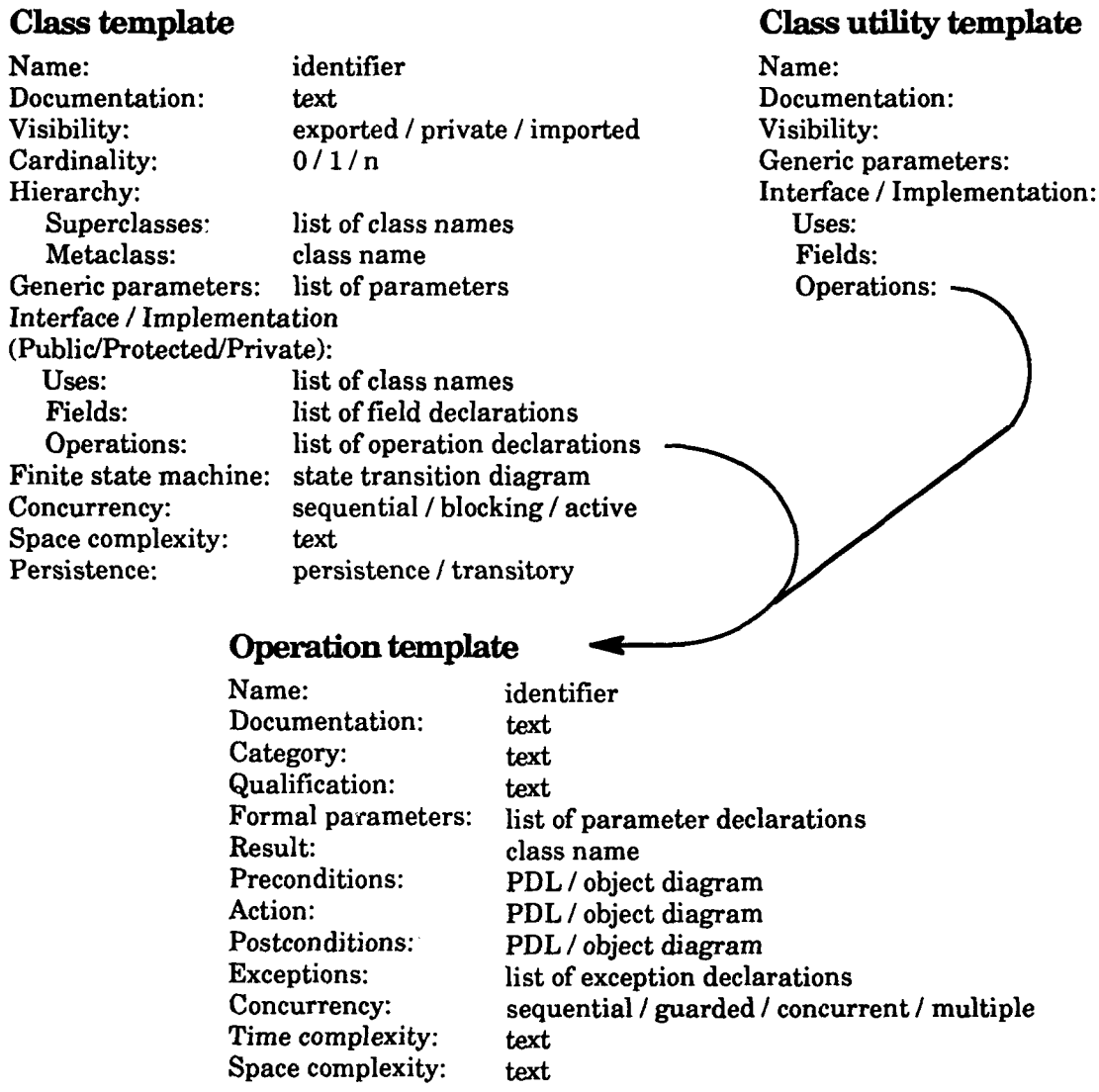


Figure 27. Class Diagram Templates [7:inside cover].

3.5.2 State Transition Diagrams. As noted in Figure 27, the finite state machine element of the class template is documented through the use of state transition diagrams. The dynamic behavior of classes is represented in Figure 28 using a slight variation of the typical state transition diagram notation. There is also an associated template to further document every state transition.

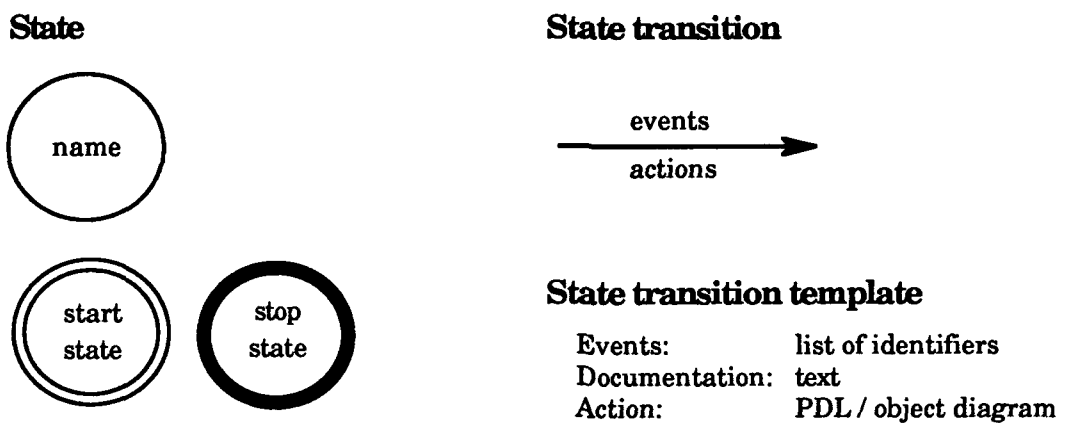


Figure 28. State Transition Diagram Icons & Template [7:inside cover].

3.5.3 Object Diagrams. The object diagram is used to semantically illustrate the operations and finite state machine dynamics of the important mechanisms that manipulate the class diagram abstractions. "Object diagrams are prototypical: each one represents the interactions that may occur among a collection of objects, no matter what specifically named instances participates in the mechanism" [7:169]. The complete object diagram notation and documentation templates are displayed in Figure 29.

Objects and their relationships are the principle components of the object diagram; however, object visibility and message synchronization add detail to the diagram. The solid line blob/cloud, unlike the class's dotted line, indicates an object instance. Note that the object's properties, especially concurrency and persistence, may be placed in the lower left "corner" of the cloud. The relationship lines indicate that messages are sent between objects both inside (solid) and outside (gray) the system.

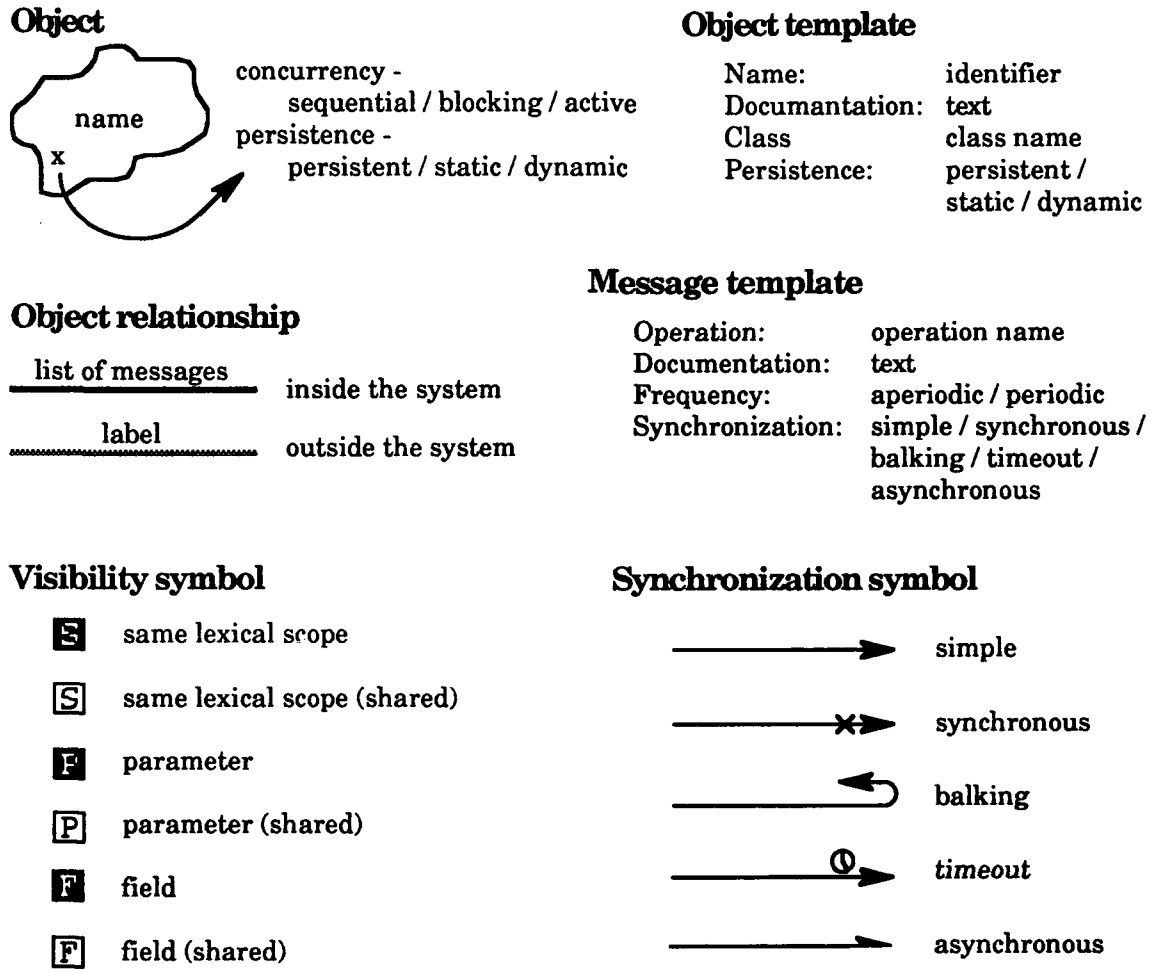


Figure 29. Object Diagram Icons & Templates [7:inside cover].

The object visibility symbols should be used (only when necessary) to indicate how two object see each other. Booch states hierarchical positioning on the diagram or nesting objects within another (to show master-slave relationships and aggregation, respectively) can aid in understanding visibility. If specific message synchronization is needed on the object diagram, the respective directed line notations of Figure 29 can be

used. These icons may also be labeled, as the object relationship lines, with a list of messages.

3.5.4 Timing Diagrams. Booch suggests three methods to express time-ordered events and the flow of control of the objects. The first method is to simply number the object messages on the object diagram in the order they are implemented. When the flow of control is not strict and orderly, but conditional, a program design language (PDL) or structured English description of the control flow can be included with the object diagrams. Finally, an adapted timing diagram (as Figure 30 illustrates) can be helpful.

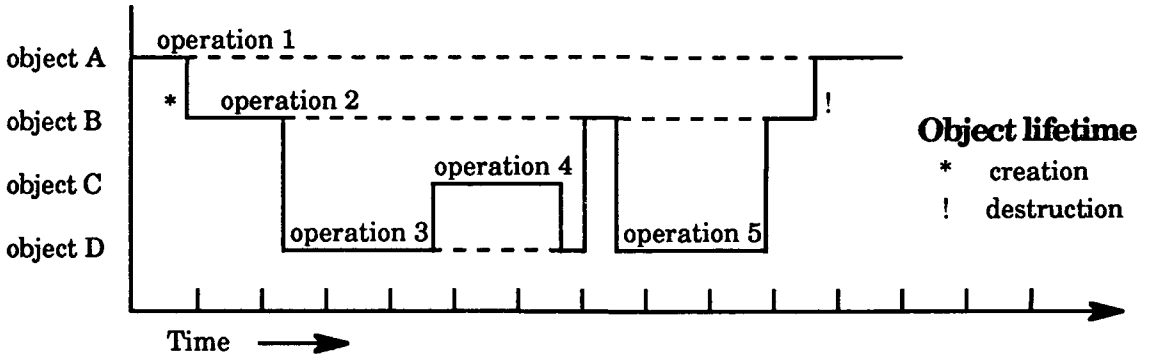


Figure 30. Timing Diagram Icon [7:inside cover].

From the notation we can see that object lifetimes can be shown on the timing diagram using creation (*) and destruction (!) symbols. Several timing diagrams may be stacked to represent multiple threads of control, or used to show primary dynamic behavior versus asynchronous external

events and/or exceptional condition flows of control [7:174]. Hard real-time constraints may also be annotated on the timing diagram.

3.5.5 Module Diagrams. As we head towards implementation module diagrams are used for the physical design of class and object allocation. Specific design allocation of the classes and objects will depend greatly on the implementation language chosen. The complete notation and template for module diagrams, including the subsystem icon for clustering parts of the system, is shown in Figure 31.

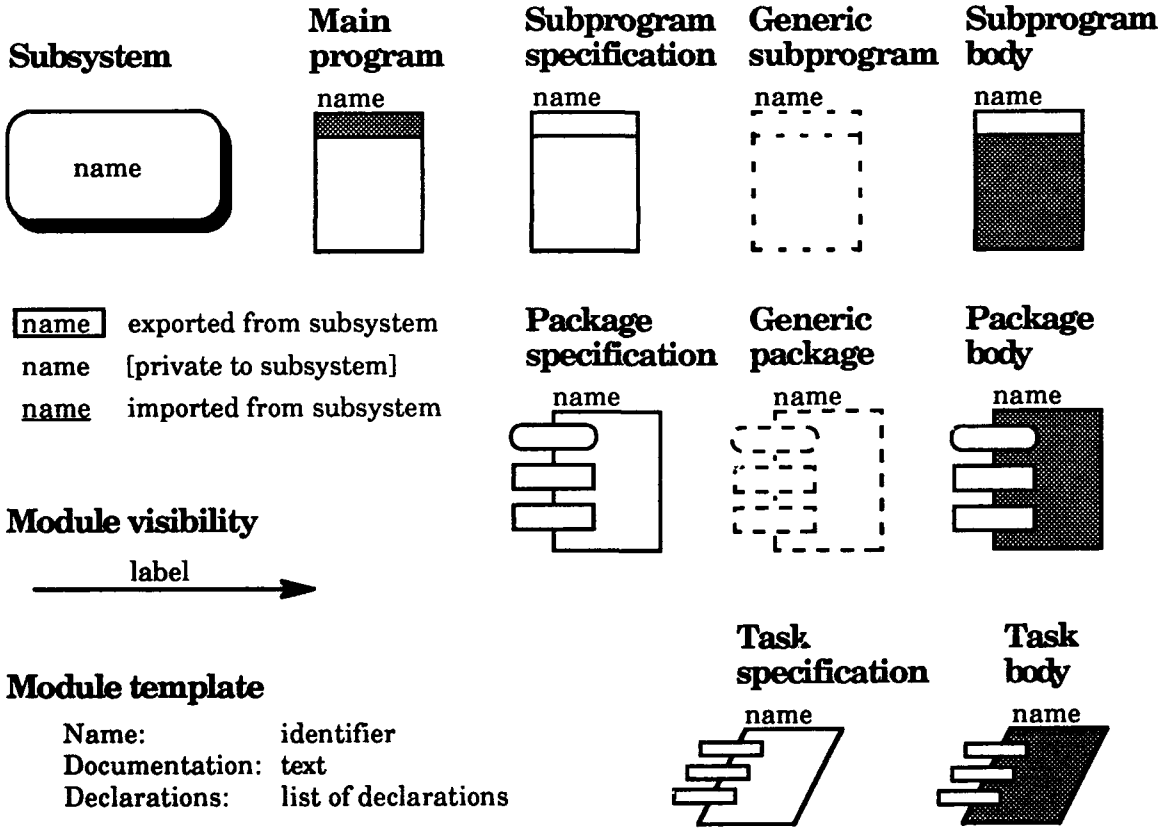


Figure 31. Module Diagram Icons and Template [7:inside cover].

Though each icon is shown separately, a module that has two parts (specification and body) would be depicted as "shadowed" specification, since the body would be below it. Note the three different ways a module can be named, depending if it is private, exported, or imported. The visibility arrow shows compilation dependency by pointing to the module(s) it (another module) depends on. Documentation of the modules can be completed using the template for static semantics, or a timing diagram/PDL for dynamic semantics.

3.5.6 Process Diagrams. When required by a large enough system that demands a distribution of the system programs, the process diagram comes into play. The icons and templates of Figure 32 are used to visualize system process allocations as well as the involvement of devices.

Connections show that the processors and devices communicate through some hardware (or radio transmission) coupling. If the connection is shown to be in just one direction, an arrow may be used. How processes are scheduled within a processor can be indicated in the lower left corner of the processor icon. Once again, templates may be used to document any further explanations.

As with any notation, Booch states that the diagrams continue to evolve during the design process as detail and decisions necessitate. He also contends that this methodology can scale up and down depending on the system being developed. Booch's language-independent approach to object-oriented design "mirrors his move from concern simply with Ada-

targeted systems to consideration of a much wider range of potential implementations" [51:109].

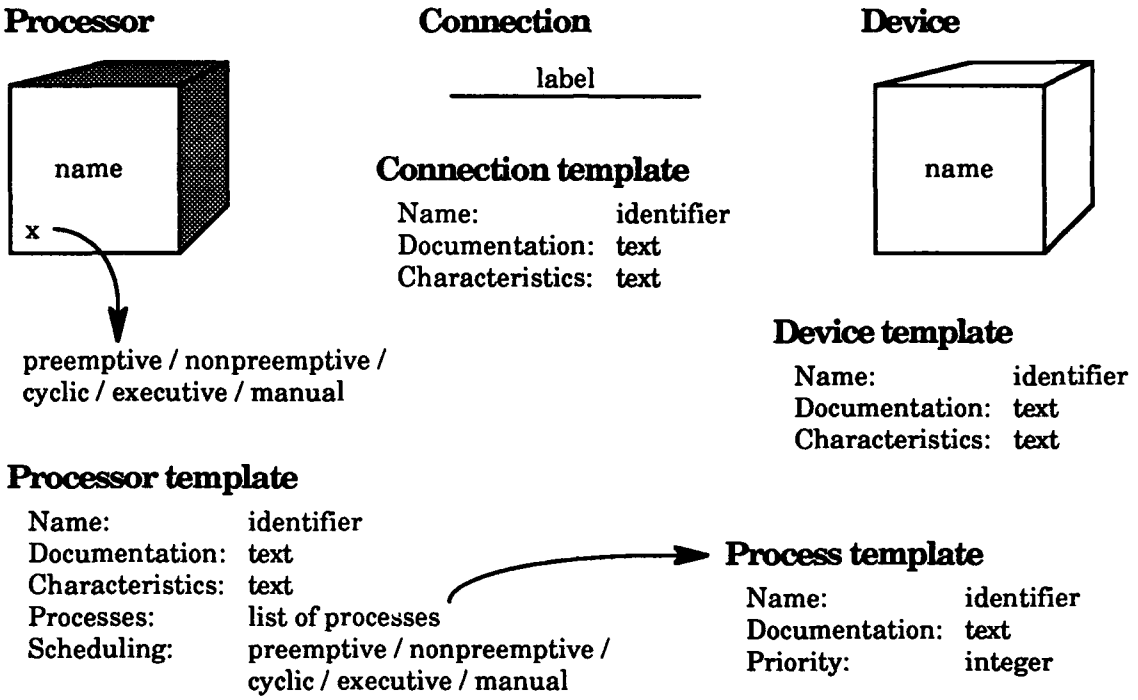


Figure 32. Process Diagram Icons and Templates [7:inside cover].

3.6 Eclectic Object-Oriented Modeling

Blending Harel's higraph and Statechart notations [26], the notation of Coleman, Hayes and Bear's Objectcharts [15], the object-oriented analysis methodology of Coleman and Hayes [29], and the "best parts of others" (such as Rumbaugh et al. [45] and Booch [7]), Dyer proposes *An Eclectic Method for Object-Oriented Database Design* [21]. We will discuss the notations that apply as we walk through each of Dyer's six steps for object-oriented database design.

1. Develop a higraph-based ER diagram, supplemented by a natural language description describing constraints, rules, and other behaviors.

2. Use the ER diagram and text description as a specification to develop a structural design for implementing the database. The design product should be a configuration diagram together with class and object templates. Check for completeness by stating consistency conditions between relations.

3. From the configuration diagram and class/object templates define a declarative functional model using templates of transition pre-conditions, post-conditions and invariants.

4. Use the configuration diagram and functional model to define a dynamic model using Statecharts or Objectcharts.

5. State reasoned arguments and test case event flows to show that the dynamic model and functional model are consistent.

6. Iterate through the steps looking for missing classes, relationships, attributes, constraints, events, etc.

3.6.1 Higraph-Based Entity-Relationship Diagrams. Step 1 of Dyer's methodology requires the use of an extended ER diagram based on Harel's higraph notation. The higraph notation "which combines notions from Euler circles, Venn diagrams and hypergraphs, and which seems to have a wide variety of applications" [26:234], augments the ER model for the object-oriented arena. The various semantics of the higraph can be viewed from the illustration of Figure 33.

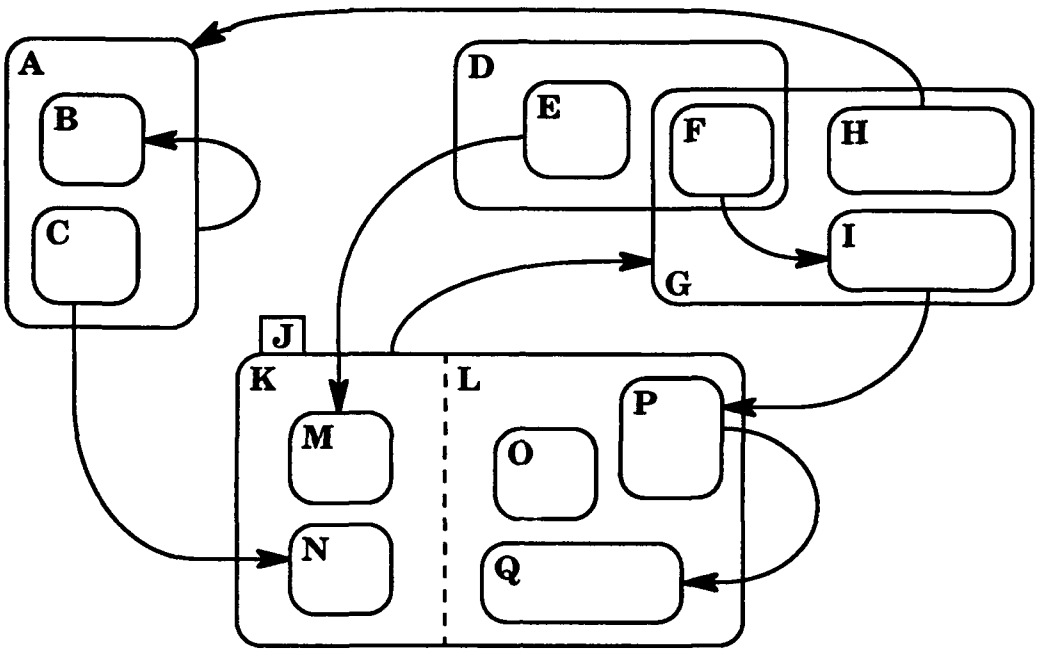


Figure 33. Representative Higraph Notation.

The notation is essentially blobs (rounded rectangles) [21:10] and arrows. Just as with Venn diagrams, sets are represented by the blobs, and blobs within a blob shows exclusive-or (XOR) subsets of a blob. Thus, sets B and C from Figure 33 are subsets of blob A. And when we look at a member of set A we see either a member of set B or set C, not one of each. Unlike the Venn diagram, however, overlapping blobs do not mean an intersection of sets unless there is a blob within the intersection. For example, the significance of the intersection of sets D and G rests solely in set F. The AND decomposition of a blob is represented by placing a dashed line between the required subcomponents of the set. Note that an element from component K and one from component L is required to make up an element of set J (labeled with a tab outside the blob). Being in J means that you are composed of a K (M or N) element and a L (O or P or Q) element.

Finally the arrows connect sets to other sets showing a relationship between elements of the sets.

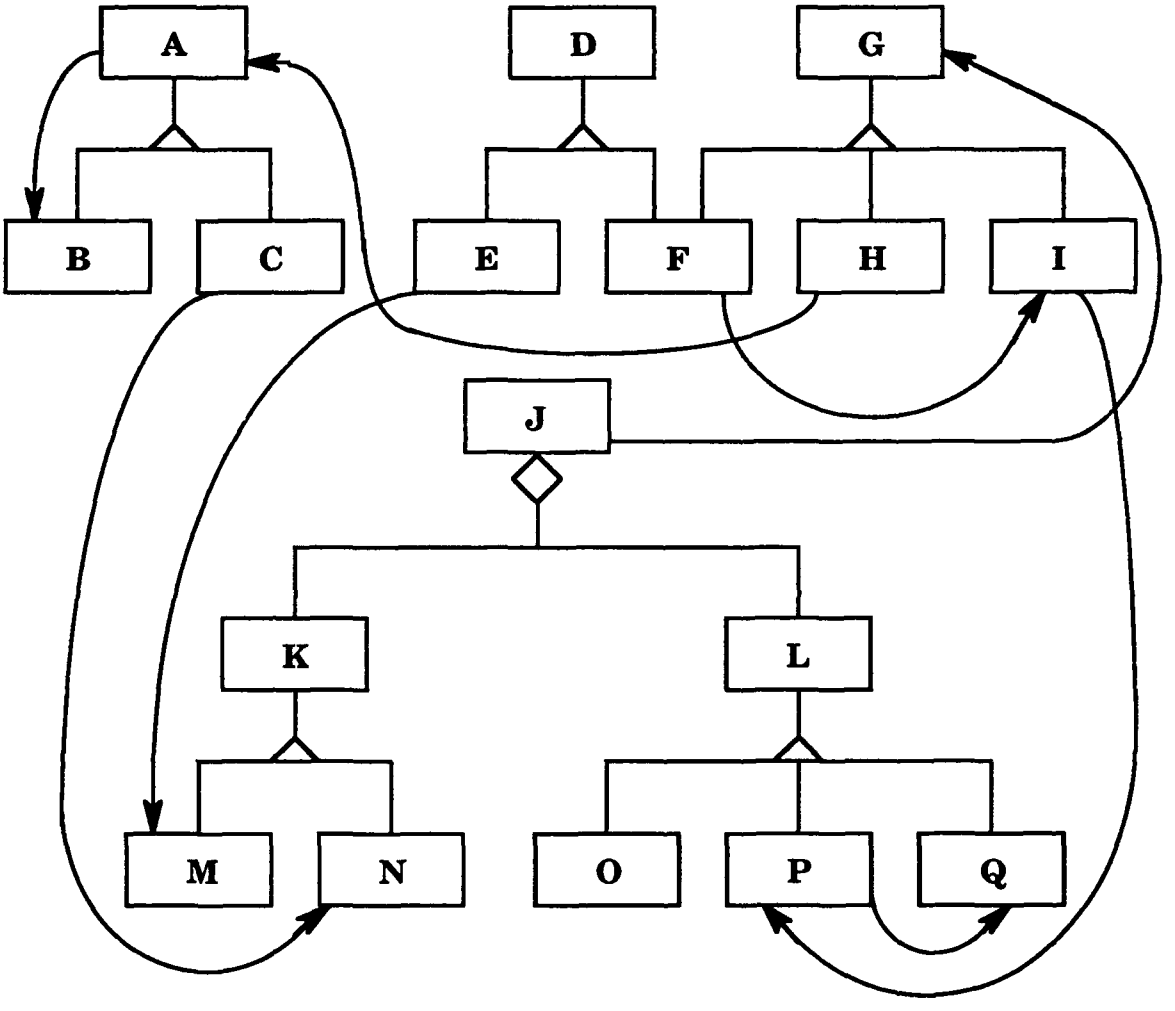


Figure 34. ER Diagram Example.

Using this notation greatly enhances the flat ER diagram by providing depth where hierarchies (is-a-kind-of and is-a-part-of) are predominant. Figure 34 shows how the compact higraph example of

Figure 33 expands when put into the extended ER diagram notation of Rumbaugh et al [45]. Both figures represent the same information, but the higraph notation of Figure 33 uses less symbols and links. The relationship arrows were left unchanged to simplify the example; however, they became longer and more easily tangled (crossed) by flattening the diagram. Notice that an unnecessary design decision had to be made with respect to blob J in order to represent the AND composition of K and L using the ER diagram. It is easy to see that the higraph notation readily allows for blowing up of smaller blobs into more detail or condensing them to eliminate clutter at higher levels of abstraction. Like the ER notations we have covered, attributes may also be listed within the blobs, and cardinality added to relationship edges.

Along with the higraph-base ER diagram, Dyer suggest using natural language descriptions to show constraints, rules, etc. The components of Step 1 result in the system specification, with which we enter Step 2.

3.6.2 Configuration Diagrams. The configuration diagram shows service interfaces between object instances. The specific notation is described by Coleman, Hayes, and Bear [15:11] using the window system alarm clock example of Figure 35. The boxes depict objects and the class to which they belong, while solid and dashed lines represent provided and required object services, respectively. Like Booch's object diagrams, the object configuration diagram shows object interfaces and which objects request services from another. They differ by not accounting for control or scheduling structures.

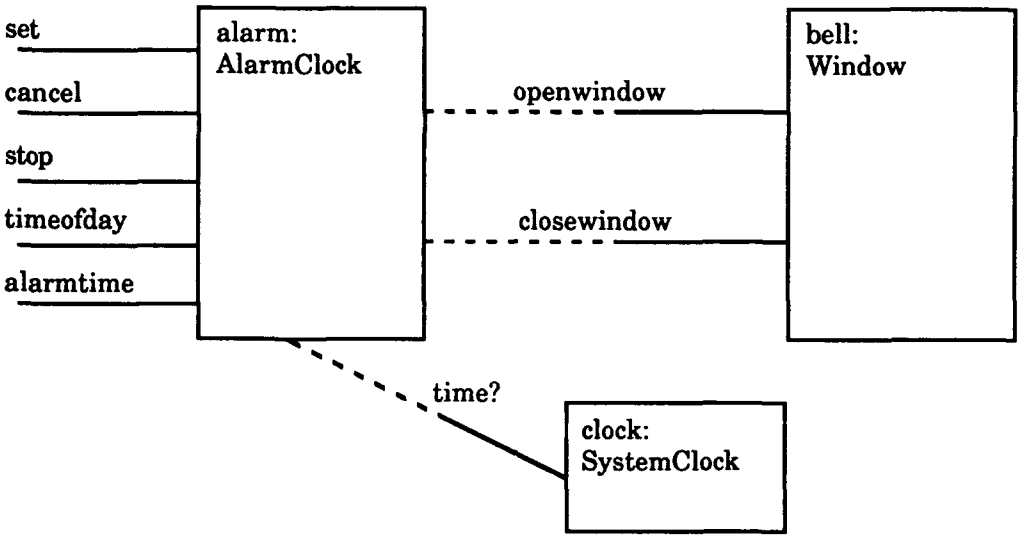


Figure 35. Alarm Clock Configuration Diagram [15:11].

Along with the configuration diagrams prescribed by Step 2, Dyer says to develop class and object templates. The recommended notation for these templates are as described by Booch, and previously represented in Figures 27 and 29 [21:23].

3.6.3 Declarative Functional Model. The functional model suggested by Dyer is the same as that presented by Hayes and Coleman [29]. They dispense with data flow diagrams of Rumbaugh et al. and describe the system level operations by pre-post condition specifications. "The resulting specifications are declarative because they do not reference how the operations are to be computed" [29:176]. The formality of the template used is not as important as having a standard to show transition pre-conditions, post-conditions and invariants.

3.6.4 Statechart / Objectchart Dynamic Model. In Step 4, Dyer suggests using an expanded form of Harel's Statecharts [26] for the dynamic model. "Objectcharts, or some other modification of Statecharts, provide the best dynamic model and also can be used easily to show consistency with our chosen functional model" [21:23]. See Figure 15 for a similar use of higraphs, to expand the dynamic model, by Rumbaugh et al. Though they will be quickly covered here, consider the appropriate reference for a detailed explanation of Harel's Statecharts [26] and the Objectcharts of Coleman et al. [15].

Statecharts are another application of higraphs. Harel defines the feature of a Statechart with the following equation:

Statecharts = state diagrams + depth + orthogonality + broadcast communication. [26:233]

Looking at Figure 36 we can see how all these features are combined. The XOR decomposition of the higraph notation reflects a state diagram, in that an object can be in only one state at a time, and allows for depth of abstraction in top-level design. For example, being in state A means that you are either in state B or C, but not both. Likewise, for higher level abstractions we may only care that we are in state A. The arrows, of course, represent transitions between states, and can have a pre-condition or guard indicated between brackets. Figure 36 shows that event *b* will only fire when D is in state G. The default arrows show that entering state Y, without specific destination states expressed, would result in entering both states B and F. Finally, the AND decomposition of components A and D

show parallel state combinations of being in state B or C while being in state E, F or G.

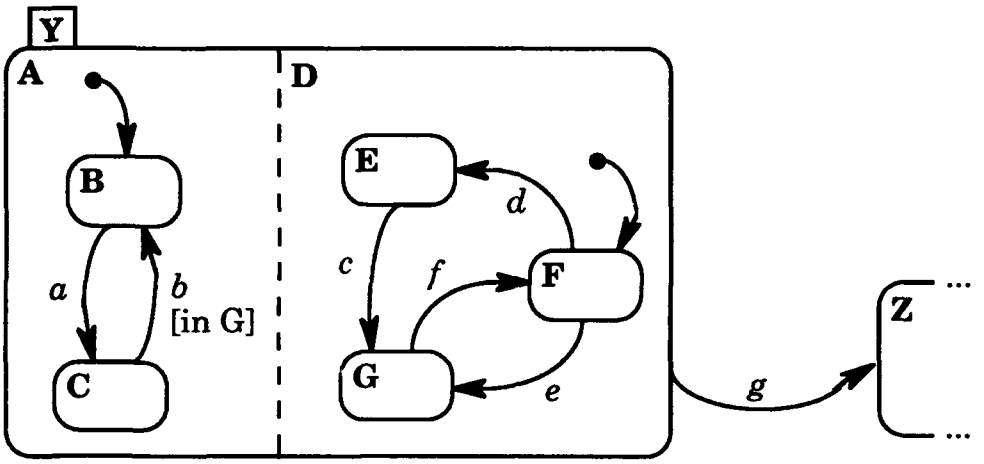


Figure 36. An Example Statechart.

Dyer contends that this notation takes "advantage of topology to reduce complexity over the flat state diagram" [21:17]. For example, transition *g* would cause a state change from any state combination of Y to state Z. This would require a transition arrow from each state of Y if we were using a flat state diagram.

The Objectcharts build upon the Statechart by specifying how transitions affect attribute values. They also differ from Statecharts by eliminating broadcast communication and replacing it with address indicators. These differences can be viewed as we compare an example Statechart (Figure 37) of the alarm clock, from the configuration diagram of Figure 35, with its corresponding Objectchart (Figure 38. Note that the *ringing* state would correspond to the *Window* of the *bell* class (from Figure

35) being open/active, and the state is obtained when the *openwindow* service is requested or broadcasted.

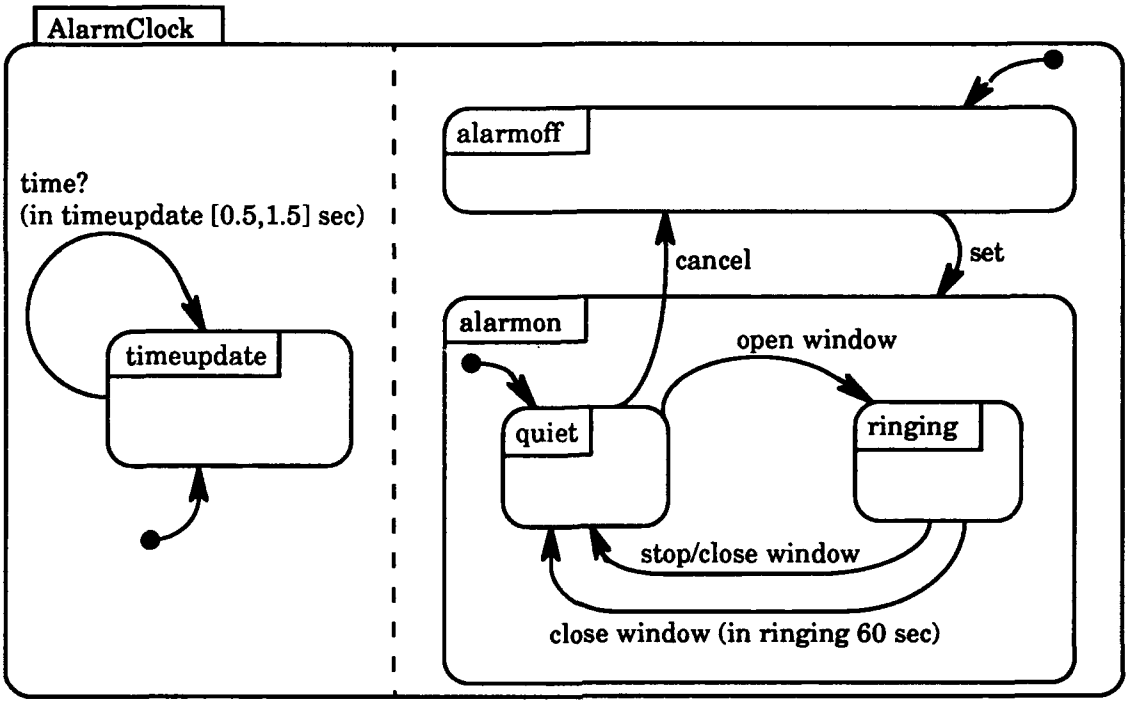


Figure 37. Alarm Clock Statechart [15:12].

Attributes that are affected by state transitions are annotated with their type in the Objectchart. For example, the attributes *timeofday* and *alarmtime* are shown within the states that they are effective. Coleman et al. define observers as services that do not change the state of an object—they may just report an attribute value [15:12]. The Objectchart enhances the Statechart by showing the name and type of allowable observers for each state. The hidden attribute, *finish*, replaces the timed transition from the ringing state. Hidden attributes are treated as observers, and are placed within brackets to show that they are internal to the state.

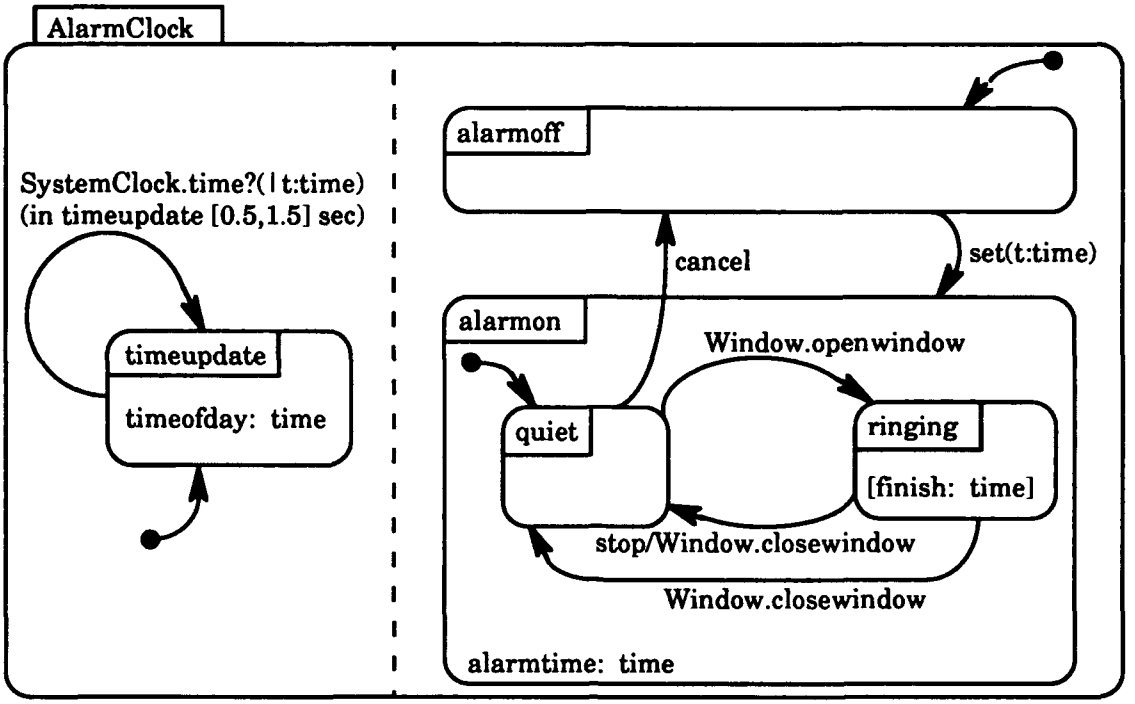


Figure 38. Alarm Clock Objectchart [15:13,16].

Services are either provided or requested by an object. Provided services are simply listed on the transition, while requested services require an address indicator of the object whose service is required. *Window.openwindow* is an example of a requested service message directed at the *Window* object. Arguments of services may be added to the Objectchart as either input or output (preceded by a "|") parameters. For example, *set(t:time)* shows the input parameter of time *t* for the *set* service, while *SystemClock.time?(| t:time)* shows the output parameter of the *time?* service.

Just as in the other object-oriented models we have seen, Dyer "concludes" his design methodology by suggesting iteration and refinement

of the design. His methodology is current and, clearly, requires more research and work as any new notation does. However, it does show the trade-offs between complexity and expressiveness that are inherent with object-oriented modeling.

3.7 Chapter Summary

We have looked at the characteristics of object-oriented data modeling and glanced briefly at four representative object-oriented modeling approaches. It was interesting to see that though most of these techniques were directed at object-oriented software development, they could easily be used for object-oriented database design (with minor tampering, if any). This outcome is natural considering the movement of application functionality into the object-oriented database arena to advocate data object behavior in a persistent environment.

In Chapter 4, we will address the promises and concerns of object-oriented technology as we analyze object-oriented information modeling versus relational data modeling. Just as promises and concerns arose when the relational database methodology developed during the hierarchic and network database generation (see chapter 2), there are many promises and natural concerns as the object-oriented technology evolves during the reign of the relational database generation.

IV. Comparisons and Analysis

4.1 Overview

As previously depicted by Figure 1, more and more of the application programmer responsibilities are moving into the database environment as we move from the function-oriented toward the object-oriented paradigm. This is an anticipated progression since, "in the object-oriented paradigm, data are considered primary and procedures are secondary; functions are associated with and attached to related data" [16:1]. Incorporating behavior with the data, and allowing for communication between data objects, naturally results in the need for a much more sophisticated DBMS. "The more sophisticated the database environment the less you, the designer, have to do" [24:50]. This statement is, of course, true for the application program designer, but not for the database designer. The redeeming quality to the increase in OODBMS design responsibilities is that the object-oriented database designer only has to do the work "once," due to the inherent reuse (of data and behavior) found in a database system. When the semantic gap is large (see Figure 2) as with the RDBMS, data is reusable, but there is much duplication of effort in the application environment to do the diverse program functions performed on the data.

As we analyze and compare object-oriented modeling to entity-relationship modeling it is important to note that we are not necessarily looking for a winner and a loser. Since we have seen from the literature that the major object-oriented modeling techniques generally incorporate

the features of entity-relationship modeling, either technique would be appropriate depending on the level of "complexity" of the enterprise being modeled. We quote "complexity" since some contend that, "object-oriented designs do not make complexity go away, but they repackage it into fewer but larger units" [16:8].

4.2 Object-Oriented Versus Entity-Relationship Databases

Before we go on to analyze object-oriented modeling compared to entity-relationship modeling it is a good idea to see the fundamental differences between the underlying object-oriented and relational database management systems. We should first realize our "comparison" is not necessarily one of apples to apples, but apples to oranges, since the object-oriented database technology is not yet mature enough to even consider for replacement of the relational DBMS [19:704]. This fact, however, does not eliminate the need to examine both database differences in theory, leaving the underlying OODBMS maturity issues for further research. The RDBMS research results of Section 2.6 demonstrated what can be accomplished through aggressive research into the fundamental levels of database technology.

Figure 39 provides an indication of the basic differences between traditional and object-oriented databases. Here we see the passive nature of the traditional database information and the object-oriented database's passive and active data used to reflect the behavior of objects.

In Table 4, Martin and Odell [41] summarize the different goals and characteristics of the classic relational database and the prevailing object-

oriented database. They also stress that, depending on the goals of the computing environment, either database may have advantages over the other.

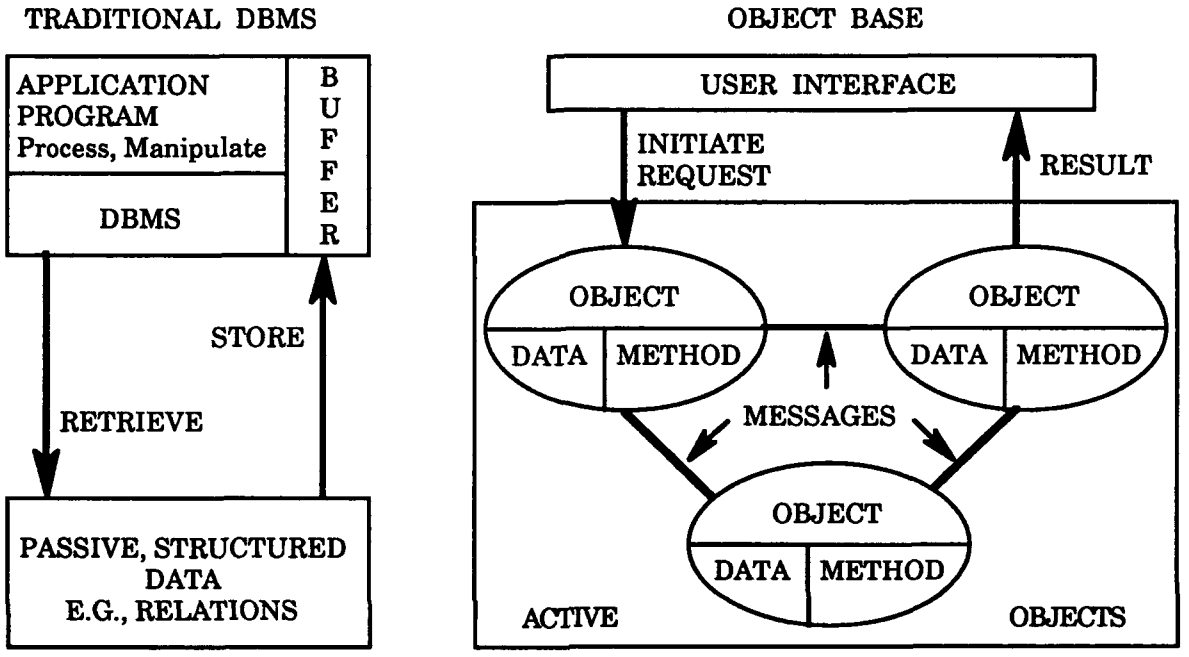


Figure 39. Traditional versus Object-Oriented Databases [42:262].

Table 4. Summary of Relational and Object-Oriented Database Differences [41:212]

Relational Databases	**Object-Oriented Databases**
Primary goal: data independence	Primary goal: encapsulation

Table 4. (Continued)

Relational Databases	Object-Oriented Databases
Data only. The database generally stores data only.	*Data plus methods.* The database stores data plus methods [services].
Data sharing. Data can be shared by any processes. Data are designed for any type of use.	*Encapsulation.* Data can be used only by methods of classes. Data are designed for use by specific methods only.
Passive data. Data are passive. Certain limited operations may be automatically triggered when the data are used.	*Active objects.* Objects are active. Requests cause objects to execute their methods. Some methods may be highly complex, for example those using rules and an inference engine.
Constant change. Processes using data constantly change.	*Classes designed for reuse.* Classes designed for high reusability rarely change.
Data independence. Data can be physically reorganized without affecting how they are used.	*Class independence.* Classes can be reorganized without affecting how they are used.

Table 4. (Continued)

Relational Databases	Object-Oriented Databases
Simplicity. Users perceive the data as columns, rows, and tables.	*Complexity.* Data structures may be complex. Users are unaware of the complexity because of encapsulation.
Separate tables. Each relation (table) is separate. JOIN commands relate data in separate tables.	*Interlinked data.* Data may be interlinked so that class methods achieve good performance. Tables are one of many data structures that may be used. BLOBs (binary large objects) are used for sound, images, video, and large unstructured bit streams.
Nonredundant data. Normalization of data is done to help eliminate redundancy in data. (It does nothing to help redundancy in application development.)	*Nonredundant methods.* Nonredundant data and methods are achieved with encapsulation and inheritance. Inheritance helps to lower redundancy in methods, and class reuse helps to lower overall redundancy in development.
SQL. The SQL language is used for the manipulation of tables.	*OO requests.* Requests cause the execution of methods. Diverse methods can be used.

Table 4. (Continued)

Relational Databases	Object-Oriented Databases
Performance. Performance is a concern with highly complex data structures.	*Class optimization.* The data from one object can be interlinked and stored together, so that they can be accessed from one position of the access mechanism. OODBs give much higher performance than relational DBs for certain applications with complex data.
Different conceptual model. The model of data structure and access represented by tables and JOINs is different from that in analysis, design, and programming. Design must be translated into relational tables, and SQL-style access.	*Consistent conceptual model.* The models used for analysis, design, programming, and database access and structure are similar. Application concepts are directly represented by classes in the OODB. The more complex the application and its data structures the more this saves time and money in application development.

The last comparison given in Table 4 was actualized by the various iterative design methodologies discussed in Chapter 3 versus the needed transition from ER diagram to relational model tables presented in Chapter 2. This unified conceptual model allows for cheaper development and

maintenance costs by reducing translations between different models. It appears that as programming languages and data management combine in a common conceptual model the object-oriented database becomes the next logical step in database evolution to handle the integration of persistent data structures and behavior (services). Figure 40 shows how the various needs not supported by traditional relational databases converged to form this new generation of database management systems.

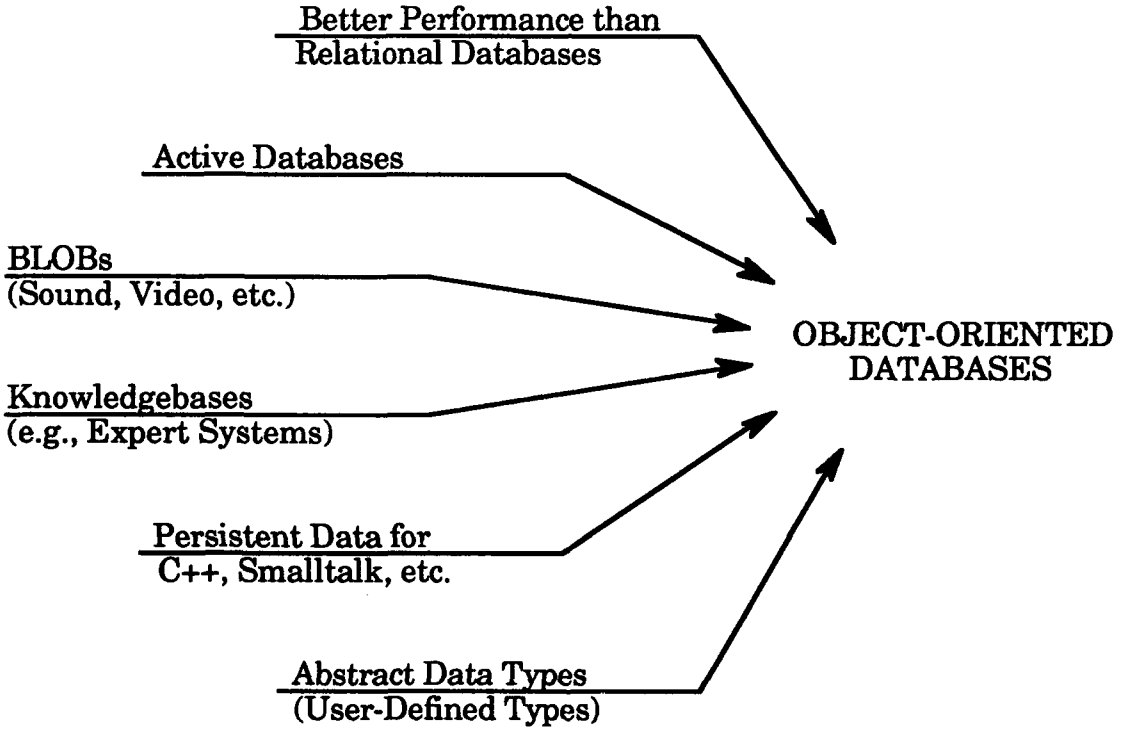


Figure 40. Converging Database Needs [41:204].

The term object-oriented as it applies to commercial databases may be misleading since "on studying marketing brochures all systems look very much alike: they claim to do everything and solve all problems" [4:3].

One classification of existing "object-oriented" databases distinguishes four types of systems from the users' perspective:

1. **Language-Oriented Database Systems**. These systems offer the user of a given language a database system adapted to that language. The goal of the system is to solve the impedance mismatch between the programming language and the database system; the data model of the database matches that of the programming language.

2. **Persistent Programming Languages**. These systems are programming language systems that offer persistent data management. There is no database system but rather a way of declaring that data is persistent or not.

3. **Engineering Databases**. These systems are database systems well adapted for engineering work. Such systems have their own data model, which is in general not object-oriented but rather entity-relationship based and supports complex object data types with attributes, hierarchical records and relationships.

4. **Object-Oriented Databases**. These systems are database systems that integrate object-oriented technology. They are true database systems supporting an object-oriented data model. [4:4-5]

Acceptance of the object-oriented paradigm (and related expenses) in the database arena will naturally bring resistance. To save past corporate investments, in the relational database, future object-oriented databases will most likely (and currently do) incorporate the relational paradigm, but at the expense of efficiency [41:213]. This fact should be taken into account when comparing performance of a relational database with that of just such an "enfeebled" object-oriented database. Another factor to consider in database comparisons is the type of queries invoked against the relational

and object-oriented databases. Relational databases are intended for queries against large sets of data while object-oriented databases implicitly assume operations against individual objects [45:329-330]. Object-oriented database navigation using pointer traversal is much faster than the relational database navigation using joins, making the relational database appear slower overall, because it is not usually optimized for single-object operations.

4.3 Object-Oriented Versus Entity-Relationship Modeling

Is there really a difference between object-oriented and entity-relationship modeling, since we have seen that conventional ER notations and constructs can be extended to include object-oriented ideas? Maybe entity-relationship modeling (with extensions) is merely a subset of, or a first step towards, object-oriented modeling. John Hughes summarizes the relative merits of entity-relationship/relational and object-oriented methods for data modeling under the four categories: data types, data integrity, schema evolution, and data manipulation [31:115]. Note that it is sometimes difficult to separate discussion of the ER and OO modeling techniques from their underlying databases since it is the database implementation that provides the needed persistence of much of the same data modeled in software development.

4.3.1 Data Type Differences. With respect to ER modeling and the RDBMS Hughes says, "by definition, attributes of normalized relations are non-decomposable and in practice they tend to have relatively simple data types (integer, real, string, data, etc.). Operations on relations are

restricted to retrieving and updating tuples identified by attribute values" [31:116]. On the other hand, OO modeling and the object-oriented database encapsulate behavior with the data structure, and these services can also be customized for specific objects. The data types of the object-oriented data model are not limited to the simple data types of the ER model and its RDBMS; however, efforts have extended the relational database to include BLOBs, nested data types, etc. Additionally, object classes may have as "attributes" other classes, and can even be passed as a message parameter to a service.

In discussing the diagramming techniques of OO and ER modeling, Martin and Odell declare, "the major difference between object schemas and ER diagrams is that ER diagrams can express attribute types that are *within* an entity type—object schemas do not" [41:461]. This is true to the extent that in object-oriented analysis any *within* attributes (physical values) are objects in their own right, thus the only real attribute types depicted are functions or associations with other *outside* objects. Later in the design process, implementation details may prompt the need to distinguish between attributes that are *within* (physical values) or *outside* (pointers) of objects. Because entity-relationship data modeling is implementation dependent (it designs for a relational database), ER diagrams express *within* associations as attribute types and *outside* associations as relationship types with other entities. Realistically, as we have seen with the object-oriented models discussed in Chapter 3, object schemas use some extended form of ER diagramming since many of the simple data type "objects" serve only as *within* attributes of an object class.

4.3.2 Data Integrity Differences. "The relational model is incapable of expressing integrity constraints with greater semantic content than straightforward referential integrity" [31:117]. Most of the time the application programs manipulating the relational database must handle such constraints (e.g., multiplicity of a relationship: one-to-one, one-to-many, etc.). If one application fails to enforce those constraints, data integrity is essentially lost. In the object-oriented model, however, services of an object enforce data integrity (only once). Services are the only means through which objects can be manipulated, thus all application programs using the database end up using the same code through those services.

Entity integrity is also different since the relational model uses the values of primary keys to uniquely identify entities, while an object is defined as having a unique identity in object-oriented modeling. A change in object state does not affect its identity.

4.3.3 Schema Evolution Differences. The ability of the data model to evolve with changing requirements defines schema evolution [31:115]. Schema evolution with the relational model has limited facilities and is restricted to the kernel of the system domain objects. Because the data structures of the database and application programs that manipulate them are loosely connected, changes to the schema requires changes to both database and applications. Since "application code" reuse is inherent in object services and tightly coupled with the data structures in object-oriented modeling, changes to the schema are considerably more feasible.

4.3.4 Data Manipulation Differences. "A fully-integrated, strongly-typed data manipulation language has significant implications for integrity preservation" [31:152]. This is obtained through the well-defined services provided by objects in object-oriented data manipulation. The impedance mismatch between SQL (and similar languages) and the application programming languages using a relational database is probably the major difference between OO and ER data manipulation. In object-oriented data manipulation the programming languages have little, if any, impedance mismatch, and work primarily through the well-defined services. However, "not every user of an object-oriented database wants to interact with the system through a procedural programming language. There is clearly a need in most application areas for a high-level query language and tools such as report generators" [31:153]. Here we can see there is still needed research in the area of object-oriented "extended" SQL, and that the relational database is not dead in areas requiring mostly set-level ad hoc queries, versus the "assumed" single object-at-a-time (actually "record-at-a-time" according to Date [19:702]) queries of an object-oriented database.

4.4 Object-Oriented Technology Benefits

Understanding the overall benefits of object-oriented technology is necessary when making a long term decision to move in that direction. Focusing on the specific areas of current immaturity in the object-oriented field may result in narrow short term business successes, but will ultimately result in the stagnation of an organization (see Section 4.5). Martin and Odell [41] provide an excellent summary of the many benefits of object-oriented analysis and design. Moreover, they conclude that the most

important benefit is the associated change from thinking like a computer (function-oriented) to making the computer "think" like a human (object-oriented) [41:41]. Many of the following benefits are realized only when object-oriented analysis and design are used with repository-based object-oriented CASE tools that generate code; however they provide a good summary of overall object-oriented benefits.

- **Reusability.** Classes are designed so that they can be reused in many systems. To maximize reuse, classes can be built so that they can be customized. A repository should be populated with an ever-growing collection of reusable classes. Class libraries are likely to grow rapidly. A preeminent goal of OO techniques is achieving massive reusability in the building of software.

- **Stability.** Classes designed for repeated reuse become stable in the same way that microprocessors and other chips become stable. Applications are built from software chips where possible.

- **The designer thinks in terms of behavior of objects not low-level detail.** Encapsulation hides the detail and makes complex classes easy to use. Classes are like black boxes; the developer uses the black box and does not look inside it. He has to understand the behavior of the black box and how to communicate with it.

- **Classes of ever-growing complexity are built.** Classes are built out of classes, which in turn are built out of classes. Just as manufactured goods are constructed from a bill of materials of existing parts and sub-assemblies, so too is software created with a bill of materials of existing well-proven classes. This enables complex software components to be built which themselves become building blocks for more complex software.

- **Reliability.** Software built from well-proven stable classes is likely to have fewer bugs than software invented from scratch.

- **New software markets.** Software companies should provide libraries of classes for specific areas, easily adapted to the needs of the using organization. The era of monolithic packages is being replaced by software that incorporates classes and encapsulated packages from many different vendors.

- **Faster design.** Applications are created from preexisting components. Many components are built so that they can be customized for a particular design. The components can be seen, customized, and interlinked on the CASE tools screen.

- **Higher-quality design.** Designs are often of higher quality because they are built from well-proven components which have been tested and polished repeatedly.

- **Integrity.** Data structures can be used only with specific methods. This is particularly important with client-server and distributed systems in which unknown users might try to access a system.

- **Easier programming.** Programs are built in small pieces each of which is generally easy to create. The programmer creates one method for one class at a time. The method changes the state of objects in ways that are usually simple when considered by themselves.

- **Easier maintenance.** The maintenance programmer usually changes one method of one class at a time. Each class performs its operations independently of other classes.

- **Inventability.** Implementors proficient with the most powerful OO-CASE tools, running on a workstation, find they can generate ideas rapidly. The tools encourage them to invent and rapidly

implement their inventions. The brilliant individual can be much more creative.

- **Dynamic lifecycle.** The target of system development often changes during implementation. OO-CASE tools make midlifecycle changes easier. This enables implementors to meet end users better, adapt to changes in the business, refine goals as the system comes into sharper focus, and constantly improve the design during implementation.

- **Refinement during construction.** Creative people such as writers and playwrights constantly change the design of their work while implementing it. This leads to much better end results. The best creative works are refined over and over again. OO-CASE tools give software builders the capability to refine the design as they implement it.

- **More realistic modeling.** OO analysis models the enterprise or application area in a way that is closer to reality than conventional analysis. The analysis translates directly into design and implementation. In conventional techniques, the paradigm changes as we go from analysis to design and from design to programming. With OO techniques, analysis, design, and implementation use the same paradigm and successively refine it.

- **Better communication between information system professionals and business people.** Business people more easily understand the OO paradigm. They think in terms of events, objects, and business policies that describe the behavior of objects. OO methodologies encourage better understanding as the end users and developers share a common model.

- **Intelligent enterprise models.** Enterprise models should describe business rules with which executives want to run their business. These should be expressed in terms of events and how events

change the state of business objects. Application designs should be derived with as much automation as possible from the business model.

- **Declarative specifications and design.** The specifications and design, built with the formality of CASE tools, should be declarative, where possible—stating explicitly what is needed. This enables the designer to think like an end user rather than to think like a computer.

- **A user-seductive screen interface.** A graphic user interface, such as the Macintosh, should be used so that the user points at icons or pop-on menu items that relate to objects. Sometimes, the user can, in effect, see an object on the screen. To see and point is easier than to remember and type.

- **Images, video and speech.** Binary large objects (BLOBs) are stored, representing images, video, speech, unformatted text, or other long bit streams. Methods such as compression or decompression, enciphering or deciphering, and presentation techniques are used with the object.

- **Design independence.** Classes are designed to be independent of platforms, hardware, and software environments. They employ requests and responses of standard formats. This enables them to be used with multiple operating systems, database managers, network managers, graphic user interfaces, and so on. The software developer does not have to worry about the environment or wait until it is specified.

- **Interoperability.** Software from many different vendors can work together. One vendor uses classes from other vendors. A standard way exists of finding classes and interacting with classes. Interoperability of software from many vendors is one of the most important goals of OO standards. Software developed

independently in separate places should be able to work together and appear as a single unit to the user.

- **Client-server computing.** In client-server systems, classes in the client software should send requests to classes in the server software and receive responses. A server class may be used by many different clients. These clients can only access server data with the class methods. Hence, the data is protected from corruption.

- **Massively distributed computing.** Worldwide networks will employ software directories of accessible objects. Object-oriented technology is the key to massively distributed computing. Classes in one machine will interact with classes elsewhere without knowing where the classes reside. They send and receive OO messages of standard format.

- **Parallel computing.** The speed of machines will be greatly enhanced by building parallel computers. Concurrent processing will take place on multiple processor chips simultaneously. (Eventually one chip will have many processors.) Objects on different processors will execute simultaneously, each acting independently. A standard Object Request Broker will enable classes on separate processors to send requests to one another.

- **A higher level of database automation.** The data structure in OO databases are linked to methods that take automatic actions. An OO database has *intelligence* built into it, in the form of methods, whereas a basic relational database does not.

- **Machine performance.** Object-oriented databases have demonstrated much higher performance than relational databases for certain applications with very complex data structures. This and concurrent computing with OO design jointly promise major leaps in machine performance. LAN

based client-server systems will employ server machines with concurrency and object-oriented databases.

- **Migration.** Existing or non-OO applications can often be preserved by fitting them with an OO wrapper, so that communication with them is by standard OO messages.

- **Better CASE tools.** CASE tools will use graphic techniques for designing classes and their interaction and for using existing objects adapted to new applications. The tools should facilitate modeling in terms of events, triggers, object states, and so on. OO-CASE tools should generate code as soon as classes are defined and allow the designer to use and test the methods created. The tools should be designed to encourage maximum creativity and continuous refinement of the design during construction.

- **Industry class libraries.** Software companies sell libraries for different application areas. Application-independent class libraries are also important and these are best provided as a facility of CASE tools.

- **Corporate class libraries.** Corporations should create their own libraries of classes that reflect their internal standards and application needs. The top-down identification of business objects is an important aspect of information engineering. [41:32-36]

Note that many of these benefits rely heavily on the use of object-oriented CASE tools with a code generator and a repository. This is because object-oriented techniques are more disciplined than conventional structured techniques, and thus require CASE technology to provide the "best way we know to build true software engineering" [41:31]. This will be especially relevant as we head toward more of a model-based software development (see Section 5.4).

Though some of the benefits listed here represent potential benefits of object-orientation, many of the benefits have already been established through studies and first-hand experience. One such study was done by Mancl and Havanas on the impact of C++ on software maintenance [40]. Though their paper addresses the different languages of C and C++ used in a major telephone company system, the actual comparison was between the different styles of object-oriented programming and conventional structured programming. They conclude from their study four interesting maintenance benefits to their company:

1. Software reuse in the maintenance of the studied system has more than doubled. They achieved 34% reuse from using their own class libraries and 19% reuse in new class construction simply through inheritance.

2. New features added to the object-oriented sections of the system required less effort and fewer interface changes within the system. OO interface changes were done 41% of the time compared to 74% of the time for the conventional structured design parts of the system. They stress that new feature development incorporates about one-half of their maintenance activity.

3. Accommodating external changes on the system resulted in fewer lines of code changed for object-oriented modules than for structured design modules. Because of the larger number of object-oriented files (using C++) the number of code blocks (contiguous lines of code) stayed about the same as for structured design modules.

4. Due to inheritance, redesign of selected subsystem using the object-oriented paradigm are expected to save further in the

maintenance effort. (They estimate 60% reduction in subsystem size for this case.) [40:69]

One area not mentioned directly in the list, but indirectly through the benefits resulting from thinking in terms of events and objects, is that of embedded systems and their derived distributed real-time system environments (as seen in the Space Shuttle's multi-computer architecture). Levi and Agrawala show that the desired methodology used in distributed real-time systems "employs *objects* as the system's elements and assigns time properties to them with *calendars,* which are data structures used to keep track of all known time events for this object in the future. This object-oriented architecture achieves objectives set for both fault containment and predictable temporal behavior" [36:5]. Though still mostly in academia, object-oriented modeling for embedded real-time systems is becoming increasingly more utilized due to the natural hierarchy of objects in the real-world [3].

The area of distributed databases was also indirectly addressed in Martin and Odell's list of object-oriented technology benefits. It is suggested "that object-oriented techniques could provide a powerful 'glue' for integrating heterogeneous systems" [6:385]. Though distributed object-oriented databases are in the infant stage "the object-oriented approach provides a particularly clean way to decompose distributed computations, treating objects as independent *agents* cooperatively computing by sending each other messages to execute associated methods" [10:199].

One last point of interest to the benefits of object-orientation that demands repeating is in the area of user-friendly interfaces (appropriately

listed earlier as "user-seductive" screen interfaces). "To date the main impetus for the development of object-oriented techniques has been in the area of graphical user interfaces (GUIs). Many of the GUIs which are so popular today, such as the Macintosh, Microsoft Windows, OSF Motif and Open Look, are object-based" [6:381]. It is our contention that much of today's commercial software capabilities are not fully utilized because of poor user interfaces. Screen appearance and overall user-friendliness of an application, in a world that is spending more and more time in front of the computer monitor, is paramount to the success of the product. When a system is easy to use, more if its capabilities are used, and thus realized as helpful to the user.

Though important as a GUI, the "user-seductive" nature of interfaces goes beyond the computer screen. Human factors engineering is playing an important role in computer science, as we endeavor to make the computer "think" and "act" naturally with humans. An up and coming area in object-based user interfaces is that of the three dimensional GUI called virtual reality. Devices such as head mounted displays and motion detecting gloves allow for user interaction with "invisible" real-world objects in an environment much like the *holodeck* of the modern television series *Star Trek: the Next Generation*. Think of the military benefits of applying current technology in this way to witness and interact with battle field simulations much as a child plays with plastic Army (Air Force) toys.

4.5 Switching to the Object-Oriented Paradigm

"The object oriented paradigm is based on an alternative way of looking at the old dichotomy of procedure and data, operator and operand; it is novel yet also a natural extension of well-established software engineering ideas" [16:2]. Though the object-oriented paradigm appears to be our next "natural extension" to make in software engineering, we need to proceed wisely. The idea of planning for new development is nothing new. Even Jesus addressed the issue two thousand years ago.

> Suppose one of you wants to build a tower. Will he not first sit down and estimate the cost to see if he has enough money to complete it? For if he lays the foundation and is not able to finish it, everyone who sees it will ridicule him, saying, "this fellow began to build and was not able to finish."
>
> [30:(Luke 14:28-30)]

Coad and Yourdon state that the object-oriented paradigm's fundamental differences from traditional methods represent a "new technology curve" rather than the latest period of time on the current structured programming "revolution" curve [13:155]. The evolutionary curve of Figure 41 describes this path that most technologies travel. They ask, if this is the best time to "jump" off the stagnated structured techniques curve and onto the object-oriented curve. We will address each of the following four questions they submit for this decision:

- Is the object-oriented paradigm sufficiently mature and well developed?
- Is there a good object-oriented implementation technology available? Does the development organization provide adequate

tools for its practitioners to effectively use object-oriented techniques?

- Is the development organization sophisticated enough to successfully change its development methods?
- Are the systems and applications being developed by the organization the kind that will most effectively use the object-oriented paradigm? [13:155-156]

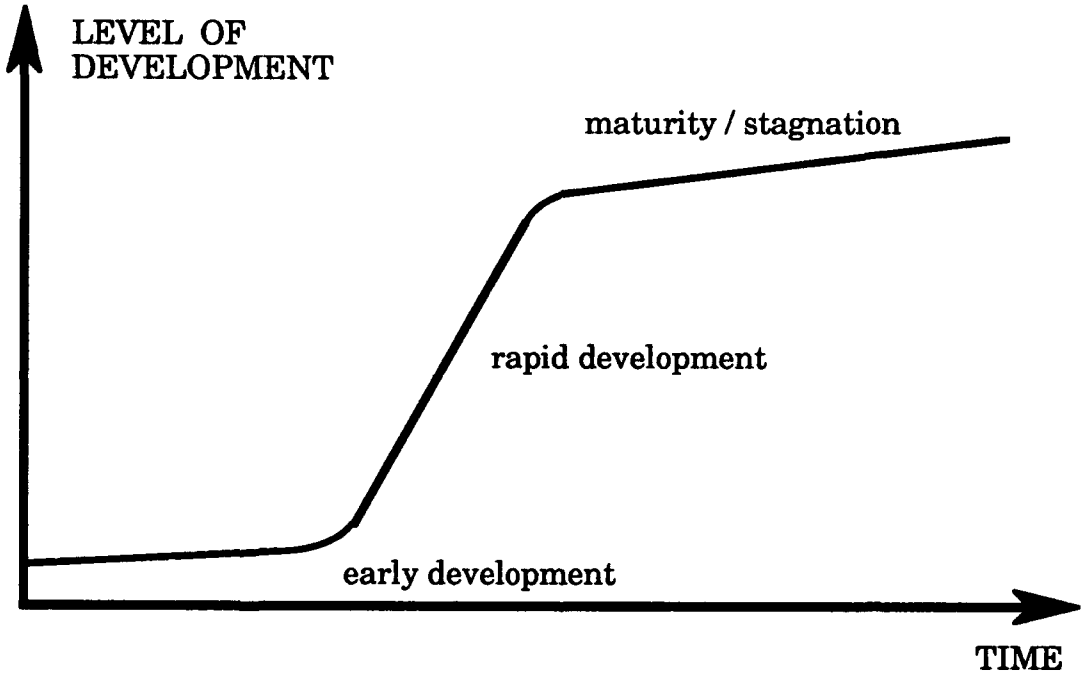


Figure 41. Technology Evolution Curve [13:155].

4.5.1 Object-Oriented Paradigm Maturity. Though the maturity level of any new technology is important to consider when choosing to adopt its techniques, this should not be the case with the object-oriented paradigm. In one sense object-oriented technology is new as it relates to

acceptance in design practice and database evolution. However, object-oriented ideas in programming and modeling are not new and, in fact, are mature as a technology. We would even consider entity-relationship modeling as a big step in object-oriented practice, since entities are abstractions of real-world objects. Object-oriented maturity and the associated risk involved "is in the eye of the beholder" and will depend ultimately on the aggressiveness of the organization.

The major issue of object-oriented maturity lies in the database management system which is key to increasing data persistence requirements. Since an OODBMS combines the well-developed object-oriented language technology with the accomplished database technologies to date, the real concern should be with the maturity of that alliance and the resulting research challenges (both hardware and software) that surface. Silberschatz et al. point out several areas where database research contributions will be needed, at the basic level, to fulfill the demands of future DBMS applications [47:114-120]. Even as the maturity of databases continues to improve and thus the risks decrease, object-oriented design is effective even when implemented using a relational, network, or hierarchical DBMS [20:22], because "the use of object-oriented design transcends the choice of database" [45:366].

4.5.2 Object-Oriented Implementation Technology. Though object-oriented programming languages do not meet all the needs of the object-oriented paradigm, they provide a richness of expressiveness, convenience, error protection, and maintainability, which are critical factors in deciding to go object-oriented. "Nevertheless, if you must use a non-object-oriented

language, you will benefit from object-oriented analysis and design, even if you lack some capabilities in the implementation language" [45:340]. Rumbaugh et al. directs an entire chapter to demonstrate how to "manually" map (versus the automatic mapping of an OO language compiler) object-oriented concepts into non-object-oriented languages (like C, Ada, and FORTRAN). They address eight required steps to implement an object-oriented design [45:340-363].

- Translate classes into data structures.
- Pass arguments to methods.
- Allocate storage for objects.
- Implement inheritance in data structures.
- Implement method resolution.
- Implement associations.
- Deal with concurrency.
- Encapsulate internal details of classes.

After describing these steps specifically for C, Ada, and FORTRAN, Rumbaugh et al. conclude that "an object-oriented design will simplify your task and provide greater flexibility and extensibility if you are willing to program in a disciplined manner" [45:362]. Object-oriented notation simply makes the conceptual mappings that must be done more explicit.

From the database standpoint of object-oriented implementation technology, current object-oriented database management systems are still in their infancy. However, there is hope that future OODBMS development

will provide the robustness expected of the current commercial relational DBMS [45:329; 47:113]. In general, OODBMSs combine the expressibility of object-oriented programming languages with the persistence of the DBMS. "OO programming languages are efficient at quickly navigating from one object to another by traversing pointers. A relational DBMS performs navigation by using joins, which are several orders of magnitude slower than pointer traversal" [45:329]. This, of course, implies the assumption that object-oriented queries are against individual objects rather than against large sets of data, as the relational database is intended to function. Regardless of the technological level (which continues to improve) of the programming languages or database management systems, we have consistently seen throughout this thesis that the object-oriented paradigm is beneficial in both software and database design.

4.5.3 Software Process Maturity. Humphrey describes five levels of process maturity for organization assessment, where "the primary objective is to achieve a controlled and measured process as the foundation for continuing improvement" [32:5]. The general characteristics of each level are reviewed as follows:

Level 1: The *Initial* Process could properly be called ad hoc, and it is often even chaotic. Until the process is under statistical control, orderly progress in process improvement is not possible. While there are many degrees of statistical control, the first step is to achieve rudimentary predictability of schedules and cost.

Level 2: The *Repeatable* Process provides control over the way the organization establishes its plans and commitments. The

organization has achieved a stable process with a repeatable level of statistical control by initiating rigorous project management of commitments, costs, schedules, and changes.

Level 3: The *Defined* Process is where the organization has achieved the foundation for major and continued progress. The organization has defined the process as a basis for consistent implementation and better understanding. At this point advanced technology can usefully be introduced.

Level 4: The *Managed* Process is where software organizations should expect to make substantial quality improvements. The organization has initiated comprehensive process measurements and analysis. This is when the most significant quality improvements begin.

Level 5: The *Optimizing* Process is when the data is available to tune the process itself. The organization now has a foundation for continuing improvement and optimization of the process. [32:5-12]

From these descriptions of the five process maturity levels we can see that "jumping" to the new technology curve of the object-oriented paradigm can only be effective when an organization is at the Defined Process Level. This also reveals an interesting consequence; the desire to effectively use the object-oriented paradigm may force organizations to improve their software process maturity (at least to level three), thus resulting in a more sophisticated development organization. Practice of object-oriented techniques may at length be an indicator of the thriving software organization.

4.5.4 Object-Oriented Applications. Unless an organization is building systems that will exploit object-oriented techniques, is there a need to make the technology curve transition to the object-oriented methodology? Constantine points out that, "while object-oriented development may have some advantages over structured development without objects, some of the claimed or demonstrated advantages are actually due to issues in software organization that are really independent of object-orientation itself" [16:9]. The truth of this statement depends, of course, on the organization's current and future products, along with the long range goals of the organization. There are the obvious advantages of using object-oriented techniques when building new systems such as graphical user interface environment applications, where "the older structured approach fails" [13:158]. However, we have seen that object-oriented technology improves overall software development and maintenance, even for systems that do not exploit object-oriented techniques. The amount of realized improvement, in this case, may not be enough for some more conservative organizations to justify welcoming the new paradigm. However, such a short term view of the object-oriented adoption costs and benefits will result in the ultimate stagnation and deterioration of an organization's growth.

4.5.5 Requirements for Object-Oriented Approach. "Indeed, object-orientation can be beneficial in building any type of application; but a more important issue is the way object-oriented approaches are promoted within an organization and incorporated to support (or perhaps even establish) a software development process" [34:92]. Konstantinow addresses some

requirements for adopting the OO approach and gives guidelines for promoting an object-oriented development environment.

Konstatinow's premise "is first that an object-oriented approach is applicable to large-scale system development and second that it actually can help in building reliable and maintainable applications" [34:94]. He contends that three things are required for the adoption of object-oriented development:

- An object-oriented design methodology,
- An object-oriented programming language, and
- An object-oriented database management system.

An object-oriented design methodology provides a common language between users and developers to communicate a real-world problem and understand system requirements in the form of real-world objects, their behavior, and the interfaces among them. There is no initial urge to functionally decompose the system into "modules," or force a data structure on an object because a particular programming language is used. Requirements can be easily validated by users and developers against specifications and design of system objects, while implementation details are transferred to later in the development process.

Though we have seen that object-oriented concepts can be mapped to non-object-oriented languages, a good object-oriented language is required "for effective implementation of a system designed according to an object-oriented paradigm" [34:96]. Konstantinow defines a good object-oriented language as one that not only supports the definition and use of classes,

inheritance for object attribute and operation definition, type definition, and constructs for procedural computation; but also supports features for data abstraction, modularity, information hiding, and reusability.

As the evolution of the database moves towards the incorporation of more and more application program responsibilities the object-oriented database management system or object management system (OMS) becomes more of a required feature for the object-oriented development process. "In object oriented development, data and procedures are closely connected; so that, in effect, the same database that is used in production to support communication among objects in the application can also be used to establish and control dependencies among the objects as they are being developed" [34:97]. This moves "management" of the data from the application programs to the database management system right from the start of development. Figure 42 shows the integrated environment an OMS can provide to an object-oriented model through the DBMS (for application objects management), application manager (including tools), and user interface to the OMS and applications.

4.5.6 Guidelines for Object-Oriented Promotion. "[Object-oriented] methodologies offer tremendous potential for realizing a major software engineering goal: building reliable applications from reusable components in a way that, in the long run, will prove to be less expensive and more efficient than traditional methods" [34:100]. Konstantinow's long term view of object-oriented technology adoption is critical in the decision to "jump" to a new technology curve. This "jump" can better be described as a smooth transition with the least amount of disruption to current developmental

practices already in place, by first "focusing on the usefulness of [the object-oriented] approaches in system design, next introduce object-oriented data modeling techniques and OMSs, and finally select an object-oriented programming language for system construction" [34:99]. His guidelines are categorized into four areas: process, methodology, tools, and management.

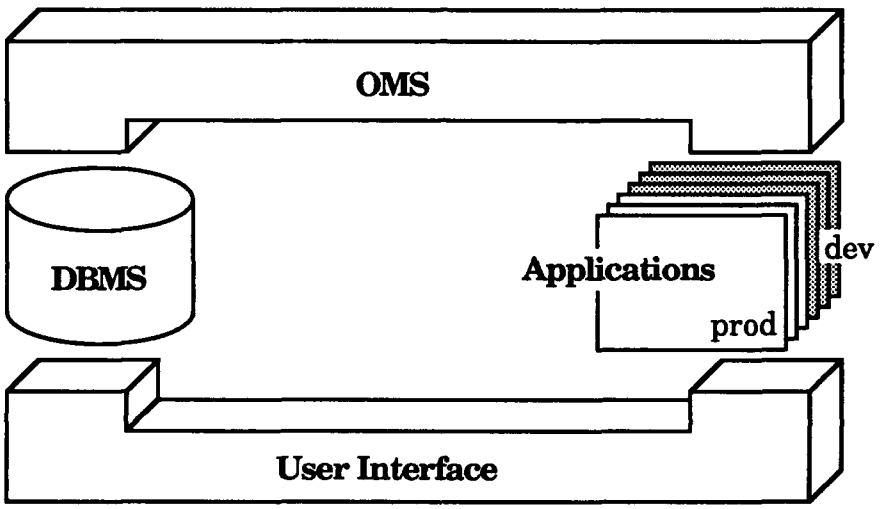


Figure 42. Integrated Object Management System [34:98].

Process

1. Outline an object-oriented development life cycle which suits the overall development environment of the organization. Match this with any practices or procedures already established, whenever possible.

2. Investigate how an object-oriented paradigm may apply to different application types (scientific, engineering, business DP) encountered in the organization. Note that even within a particular application area, the applicability may change. For

example, a compute-intensive scientific application based on a few deterministic formulas may not lend itself to object-oriented design; however, a large-scale simulation with autonomous but intricately connected communicating agents [as in war gaming] certainly could benefit greatly from an object-oriented approach.

3. Demonstrate how an object-oriented paradigm applies well to the different functional, behavioral, and data-oriented representations of [a] system.

Methodology

1. Demonstrate the technical utility of an OMS. Show how the constructs of an OMS both support object-oriented design and provide analogs or connections to other standard database models and information management systems within the organization.

2. Set forth requirements for object-oriented programming support to extend the functions of an OMS in system development. Define the minimum requirements (based on the organization's information structure) for a data definition language (DDL), a data manipulation language (DML) within the OMS, an object-oriented procedural language, and interfaces to standard 3GLs and 4GLs (including DDLs and DMLs) used in the organization. Ask what features such languages should have and how they relate to other programming methodologies adopted.

3. Analyze requirements for a flexible user interface to the OMS and development systems. New methodologies for object management, prototyping, and the like may require different presentations to developers and different database access mechanisms than are normally encountered in the organization.

Tools

1. Establish guidelines for evaluating toolsets which integrate object-oriented programming languages with OMSs. Recommend selection of an OMS with languages already integrated (as specified above).

2. Evaluate tools which offer programming extensions to 3GLs and 4GLs already accepted within the organization. The purpose is to ease the transition from standard development techniques to an object-oriented approach; in some cases, it may even be appropriate to introduce object-oriented constructs through standard programming languages [as previously discussed in section 4.5.2].

3. Plan for computing system, networking, and workstation support before introducing an integrated OMS. Delineate any expected differences between installing an OMS and a standard DBMS (along with its accompanying support environment) for use in a development environment. Show how OMS administration corresponds to or varies from database administration for DBMSs already in use within the organization.

Management

1. Demonstrate the capability and usefulness of an integrated OMS for coordinating technical activities (design, programming, and database) in system development.

2. Promote a system development plan which incorporates an integrated OMS for managing reusable objects and applications. Note that designing for reusability calls for special considerations concerning the development life cycle, identification and management of reusable components, and strong underlying database support which can be tied in naturally with the OMS. The emphasis here is that design for reusability is not just a

technical issue but has special significance for software management as well.

3. Formulate a strategic plan for managing information for all enterprises of the organization on the basis of object-oriented systems. [34:99-100]

4.6 Chapter Summary

We have seen the differences between the relational and object-oriented databases, and have discussed the relative merits of relational and object-oriented data modeling. Our concern was not to extol one paradigm above the other since each model serves the needs of different generations in software engineering evolution. We then looked at the overall benefits realized and promised from the next generation object-oriented technology when adopted fully with OO-CASE tool support. Finally we looked at the organizational concerns and requirements to effect a move to the object-oriented paradigm.

Chapter 5 concludes with the inevitability of the object-oriented paradigm in software engineering and database technology and recommends research approaches that need to be taken to benefit now and in the future.

V. *Conclusions and Recommendations*

5.1 Overview

In this chapter, we will quickly summarize the object-oriented modeling technology, which is essentially a superset of entity-relationship modeling. Conclusions drawn from the analysis of "both" modeling techniques and their underlying database management systems will then be presented for STARS evaluation. Finally, motivation and general recommendations for future research and military applications will be presented.

5.2 Summary

5.2.1 Summary of Object-Oriented Modeling. Martin and Odell's [41] pyramid illustration of Figure 43 provides an excellent summary of object-oriented modeling. Chapter 3 described several of the current modeling techniques proposed, yet all of these essentially are concerned with modeling object structure and object behavior, as shown here.

Using the terminology of Figure 43, a high-level model is built of the entire enterprise, it is then extended into a more detailed model of a particular business area, and finally built into a design for one system before actual construction. Object structure analysis concerns itself with object types, object associations, generalization, and composition; while object structure design concentrates on class identification, inheritance,

and data structure. Object behavior analysis lies with event types, states, trigger rules, control conditions, and operations; while object behavior design deals with operation identification and method design. Various diagramming methods were shown for object-oriented modeling and may never fully standardize on one method due to personal preference; however, it is recommended that OO-CASE tool builders standardize on diagrams that are already widely understood to avoid incompatibility [41:121].

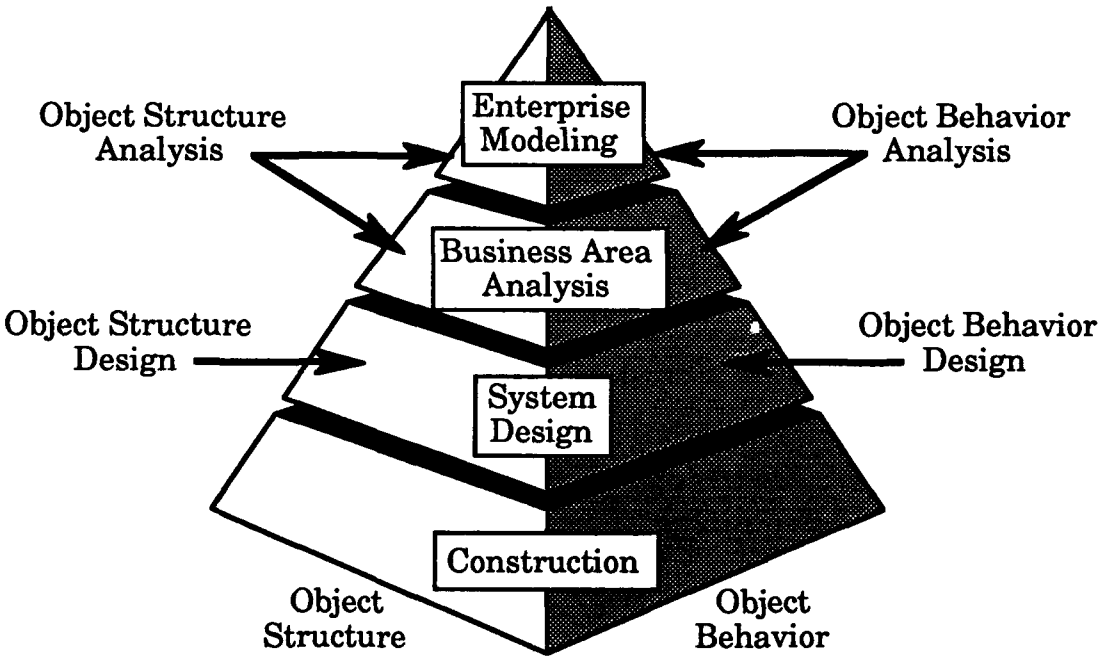


Figure 43. Interrelated Object Structure and Behavior Model [41:69].

Figure 44 shows how object-oriented modeling for software and database uses the same conceptual model for analysis, design, and implementation. "OO techniques tear down the conceptual walls between

[conventional] analysis, design, programming (or code generation) and database definition and access" [41:205].

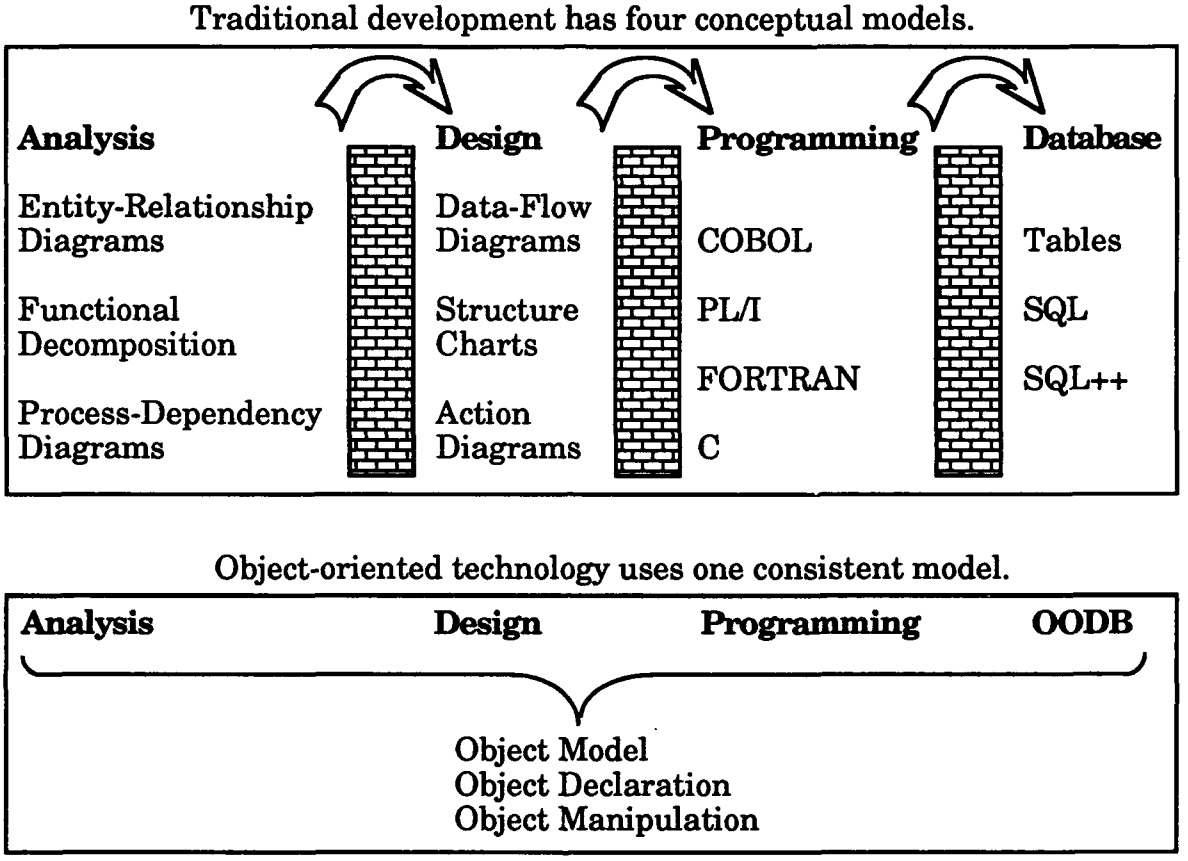


Figure 44. Object-Oriented Unified Conceptual Model [41:205].

5.2.2 Summary of Object-Oriented Design Steps. A simple summary of object-oriented design is Hartrum's [28] "Easy Guide to OOD (Based on Rumbaugh et al. [45])." These four "bits" of advice represent the basic steps associated with common object-oriented designs.

0000. Start with OOA.

- Information (ER) diagrams: objects, attributes, relationships.
- Dynamic (state) diagram: for each "interesting" object.
- Functional (DFD) diagram and process descriptions.

0001. System design/context analysis.

- Define system boundary.
- Draw context diagram, establish interfaces.

0010. Move all objects and attributes from OOA to the design. Do not add system-generated unique ID attributes.

0011. Add (if needed) a software object for each external interface (e.g. fan, sensor, and user interface for a temperature controller).

0100. Apply generalization to create new superclasses where appropriate.

0101. Add lower level objects where appropriate (should evolve over the course of the design).

0110. For each relationship in OOA (except 'ISA'), determine how to represent it (design decision). Use handle (pointer) attributes.

0111. For each object define access operations.

1000. For each object's state diagram, determine the external events that cause state transitions. Define an operation for each of those.

1001. For each operation, using the functional (DFD) diagrams, write pseudo-code. Define new operations for the associated object and for other objects as needed.

1010. Define your communication convention. For local (same object), private operations, procedure calls are appropriate.

For communication between objects, use procedure calls or messages, but be consistent. For messages, clearly specify each message type, its destination object class, and the class' operation that will be invoked. In your pseudo-code clearly indicate where messages are sent, what message type, and if a reply is awaited (blocking send).

1011. Tie it all together with a main program unit or user interface/menu as appropriate.

1100. If appropriate, apply inheritance to move common attributes and operations up the structure.

1101. Resolve inheritance.
- Leave as an implementation issue.
- Design "uninheritance" if no language support.

1110. Check for consistency.

- For each object, external operations should be clearly marked and associated with a message type or procedure calling signature.
- When invoking an operation in another object, clearly show message send or procedure call. Syntax must match that in the called object.
- Every external non-access operation should be invoked from somewhere in your design.

1111. From 1110, draw object communication diagram (object visibility). Be sure your syntax is defined.

5.3 Conclusions

Object-oriented modeling with respect to software and database development is, as the title of Rush Limbaugh's first book states, "The Way

Things Ought to Be" [37]. Though the object-oriented database management system may still be considered a risky technology to adopt due to its "immaturity," object-oriented modeling is a no-risk, high-yield investment for the long term. As we wait for OODBMSs to mature, "object-oriented concepts provide an excellent basis for modeling [the] hierarchical, network, relational, and object-oriented DBMS" [45:388]. Patrick Barnes stated "an observation [he] made during the course of [his] research is that methodologies which are language independent seem to have the most chance of surviving and being used over the long haul, [while] the more complex the method, the less it seems to be used" [5:Sec 6,3]. In the course of this thesis the object-oriented paradigm was shown to be implementation language (and database) independent as well as a simplifying, natural method for modeling real-world applications. The object-oriented paradigm should be aggressively pursued, if only to provide a common language for clear and natural communication between client and software engineer in developing complex systems. In fact, "the object-oriented style [of programming], is best suited to the broadest set of applications, namely, industrial-strength software in which complexity is the dominant issue" [7:38].

"The object model...provides a consistent mode of representation across all programming problems. [Therefore,] the object-oriented paradigm is capable of representing any program" [8:Sec 7,2-3]. Here lies the chance for standardization within the software engineering field. Object-oriented modeling represents a standard technique that can (and should) be use throughout the Department of Defense (and beyond) for both application and database development. The success of the object-oriented

paradigm in the application environment can (and will) be enjoyed in the DBMS arena as the two areas continue to evolve in codependency. The challenges of the future will demand such a continuing merger of data and behavior, as we stop thinking like computers and proceed to make our computers "think" like us.

5.4 Recommendations

In looking to the future—as opposed to looking short term to maintain the status quo in computer technology—it is only natural to dream of what may someday be. Science fiction has become more and more popular as those, once fictional, dreams of the authors continue to become fact. As mentioned earlier, the *holodeck* of television's *Star Trek: the Next Generation* is conceptually becoming a reality that has many military and commercial application potentials. For example, war gaming and medical training would be conducted using computer-generated "real-world" objects.

As Figure 45 extends the previous database evolution example of Figure 1 in the same direction we see the possibility of computer systems that "program" themselves to meet the spoken or diagrammed demands of the user—again, much like the computer systems of *Star Trek: the Next Generation*. Though still a science fiction dream, the conceptual ingredients for just such a capability are here today in the form of code generators (for self-programming), formal methods (for requirements specification), voice recognition and CASE tools (for input), etc.

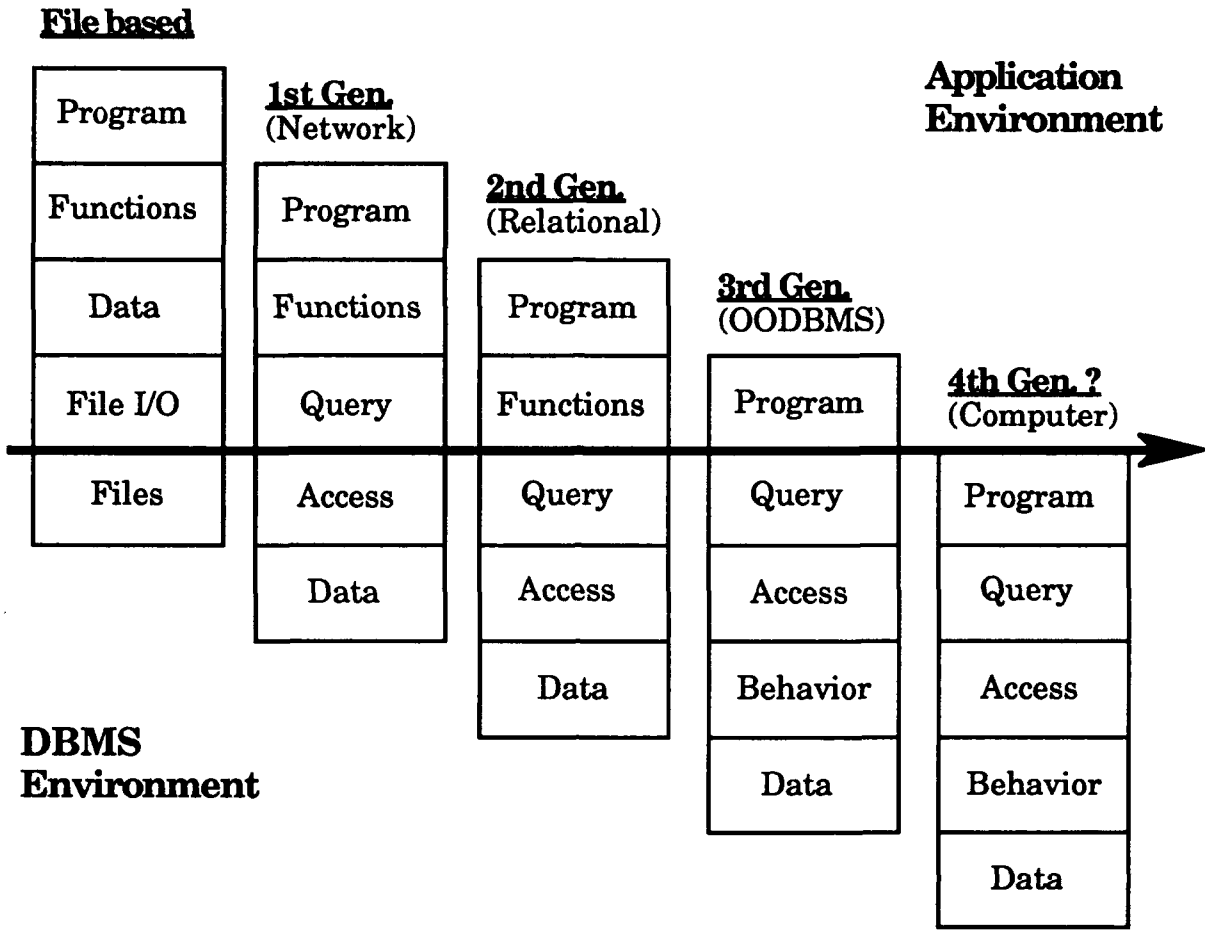


Figure 45. DBMS and Application Environments: the Next Generation.

We are headed toward model-based software development, where the model becomes the working-level user "code" and the system automatically generates the machine code without the user having to deal with mid-levels of abstraction such as Ada, C++, or FORTRAN. By modifying the model to fix problems, instead of the program, we move toward reuse and portability of design instead of code [3]. Remember that the relational database was originally considered a "toy" before it became the most popular DBMS in use

today. Therefore, STARS's trek for "the next generation of databases calls for continued research into the foundations of database systems, in the expectation that other such useful 'toys' will emerge" [47:113].

5.5 Closing Remarks

This thesis effort was essentially a strategic literature review of object-oriented modeling at two stages of complexity and a comparison of their underlying relational and object-oriented database management systems. Much of the work was done by the many experts cited throughout this thesis; however, bringing those views together enforces the notion that object-oriented modeling is not a threat to software engineering simply because it involves change. On the contrary, object-orientation was shown to be only beneficial, and revealed as the inevitable direction that software engineering is headed. The decision for any organization to embrace the object-oriented paradigm comes down to whether they wish to be on the cutting edge of software engineering or not.

Appendix A. *Glossary of Terms*

Abstract Data Type (ADT)

A type of object that contains the definition of its data structure and permitted operations. ADTs give objects a public interface through its permitted operations. However, the representations and methods of these interfaces are private. [41:451]

(1) An abstract data type consists of operations, (atomic) objects and relationships among the objects. Algebraic specifications view the operations (called by some authors the interface procedures) in terms of pure functions. These are related to one another through standard relations. [23:G1]

(2) A data structure (strictly, class of data structures) described by the set of available services or operations defined on the data structures; formally: a pair (D,P) consisting of a set, D, of logically-exported domains plus a set, P, of logically-exported operations on that domain; sometimes the programming module that implements an abstract data type. [23:G1]

Abstraction

An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer. [7:39]

ANSI/SPARC

(Acronym) American National Standards Institute / Systems Planning and Requirements Committee.

Attribute

A named property of a class describing a data value held by each object of the class. [45:455] (See instance variable)

An identifiable association between an object and some other object or set of objects. Each attribute is an instance of an attribute type. [41:451]

BLOB

(Acronym) binary large object.

Cardinality

A constraint on the number of objects that must participate in the mapping of a function. This constraint is typically expressed as a minimum and maximum number. The minimum cardinality constraint indicates the least number of objects to which a given object must map. The maximum indicates the greatest number of objects to which a given object must map. [41:452] (See multiplicity)

CASE

(Acronym) computer-aided software engineering.

Class

A description of a group of objects with similar properties, common behavior, common relationships, and common semantics. [45:455]

Conceptual View

The perspective of an entire enterprise. A high level model.

Concurrency

Concurrency is the property that distinguishes an active object from one that is not active. [7:66]

Cyclic Scheduling

Control passes from one process to another, and each process is given a fixed amount of processing time, usually called a frame; processes may be allocated time in frames or subframes. [7:183]

Data Abstraction

Abstracting common features or properties of a class of data, in particular, shared attributes, common rules to which examples of the data conform, and the operations defined on or by the data. [16:2]

Data Independence

The ability to modify how a data object is defined without affecting the application program.

The ability to modify a scheme definition in one level [of abstraction] without affecting a scheme definition in the next higher level. [35:12]

DBMS

(Acronym) database management system.

DDL

(Acronym) data definition language.

DFD

(Acronym) data flow diagram.

DML

(Acronym) data manipulation language.

Encapsulation

Encapsulation is the process of hiding all of the details of an object that do not contribute to its essential characteristics. [7:46]

The programming language realization of information hiding, the means by which the hiding of design and implementation details is enforced in the program as written. [16:2]

A protective encasement that hides the implementation details of an object, making its data accessible only by operations put there to mediate its access. It is often used interchangeably with *information hiding*. [41:453]

ER

(Acronym) entity-relationship.

Executive Scheduling

Some algorithm controls process scheduling. [7:183]

GUI

(Acronym) graphical user interface (pronounced "gooey").

Hierarchy

Hierarchy is a ranking or ordering of abstractions. [7:54]

Impedance Mismatch

Where information must pass between two languages that are semantically and structurally different, such as a declarative data sublanguage and an imperative general-purpose language. [39:206]

Information Hiding

A broad principle of software engineering wherein implementation details and the consequences of design decisions are hidden within well-defined programming packages or units. [16:2]

Inheritance

A relationship between object classes by which features of one object class become defined for another, descendent class. [16:2]

Instance

An object described by a class. [45:458]

Instance Variable

An attribute or local variable of an object class; part of the state of an object. [23:G17] (See attribute)

Instantiation

The process of creating instances from classes. [45:459]

Object creation. [41:455]

The process of creating an object instance (physical data realization) at run time; also sometimes the object instance itself. [23:G17]

LAN

(Acronym) local area network.

Manual Scheduling

Processes are scheduled by a user outside of the system. [7:183]

Modularity

Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. [7:52]

Multiple Inheritance

A type of inheritance that permits a class to have more than one superclass and to inherit features from all ancestors. [45:459]

Multiplicity

The number of instances of one class that may relate to a single instance of an associated class. [45:459]

Nonpreemptive Scheduling

The current process continues to execute until it relinquishes control. [7:183]

Object

A concept, abstraction, or thing with crisp boundaries and meanings for the problem at hand; an instance of a class. [45:460] (See instance)

A data abstraction encapsulated with the associated operations such that information hiding is supported by restricting access to data and internal state only by way of interfaces of the associated operations. [16:2]

Object-Oriented

A software development strategy that organizes software as a collection of objects that contain both data structure and behavior. Abbreviated *OO*. [45:460]

OMS

(Acronym) object management system. See OODBMS.

OMT

(Acronym) Object Modeling Technique.

OO

(Acronym) object-oriented.

OOA

(Acronym) object-oriented analysis.

OO-CASE

(Acronym) object-oriented computer-aided software engineering.

OOD

(Acronym) object-oriented design.

OODBMS

(Acronym) object-oriented database management system. See OMS.

Orthogonality

Independence between systems.

PDL

(Acronym) program design language.

Persistence

Persistence is the property of an object through which its existence transcends time (i.e. the object continues to exist after its creator ceases to exist) and/or space (i.e. the object's location moves from the address space in which it was created). [7:70]

Persistent Object

An object that survives the invoked operation that creates it. [41:456] (See persistence)

Preemptive Scheduling

Higher priority processes that are ready to execute may preempt lower priority ones that are currently executing; typically processes with equal priority are given a time slice in which to execute, so that computational resources are fairly distributed. [7:183]

RDBMS

(Acronym) relational database management system.

Referential Integrity

(In a relational database) A property of a database such that each foreign key is consistent with its corresponding primary key. [45:462]

Ensuring a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. [35:151]

RM/T

(Acronym) relational model / Tasmania.

Schema Evolution

The ability of the data model to evolve with changing requirements. [31:115]

Semantic Gap

The amount of abstraction that exists between the application semantics (real-world problem view) and the logical schema (representation) of the DBMS.

STARS

(Acronym) Software Technology for Adaptable Reliable Systems.

SQL

(Acronym) structured query language. A standard language for interacting with a RDBMS.

Typing

Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways. [7:59]

Bibliography

1. Atkinson, Malcom, François Bancilhon, David DeWitt, Klaus Dittrick, David Maier, and Stanley Zdonik. "The Object-Oriented Database System Manifesto." in *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*. Kyoto, Japan, December 1989.

2. Austin, Kenneth A., Gerald R. Morris, Nealon F. Smith, and Thomas C. Hartrum. "An Entity-Relationship Modeling Approach to $IDEF_0$ Syntax." in *Proceedings of IEEE 1990 National Aerospace and Electronics Conference (NAECON 1990)*, vol. 2. Dayton OH, May 1990, pp. 641-645.

3. Bailor, Paul. *Class lecture notes taken in CSCE 693, Principles of Embedded Software*. Fall 1992, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH.

4. Bancilhon, François. "A Classification of Object-Oriented Database Systems." in *Database Programming Languages: Bulk Types & Persistent Data*. Kanellakis, Paris and Joachim W. Schmidt, Morgan Kaufmann Publishers, San Mateo, California, 1992, pp. 3-6.

5. Barnes, Patrick Denis. "A Decision-Based Methodology for Object Oriented-Design." Master's thesis, AFIT/GCS/ENG/88D-1, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988, (AD-A202 579).

6. Bell, David and Jane Grimson. *Distributed Database Systems*, International Computer Science Series. Wokingham, England: Addison-Wesley Publishing Company, 1992.

7. Booch, Grady. *Object-Oriented Design with Applications*. Redwood City, California: The Benjamin/Cummings Publishing Company, 1991.

8. Bralick, William A., Jr. "An Examination of the Theoretical Foundations of the Object-Oriented Paradigm." Master's thesis, AFIT/GCS/MA/88M-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1988, (AD-A194 879).

9. Brooks, Frederick P., Jr. "No Silver Bullet: Essence and Accidents of Software Engineering." *IEEE Computer*, vol. 20, no. 4, pp. 1069-1076, April 1987.

10. Cattell, R.G.G. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1991.

11. Chen, Peter Pin-Shan. "The Entity-Relationship Model: Toward a Unified View of Data." *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9-36, March 1976.

12. Coad, Peter and Edward Yourdon. *Object-Oriented Analysis*, 2nd Edition. Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

13. Coad, Peter and Edward Yourdon. *Object-Oriented Design*. Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

14. Codd, E. F. *The Relational Model for Database Management: Version 2*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.

15. Coleman, Derek, Fiona Hayes, and Stephen Bear. "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design." *IEEE Transactions on Software Engineering*, vol. 18, no. 1, pp. pp. 9-18, January 1992.

16. Constantine, Larry L. "Object-Oriented and Function-Oriented Software Structure." in *Object-Oriented System Symposium by Digital Consulting, Inc.*. December 1990.

17. Data, C. J. *Relational Database: Selected Writings*. Reading Massachusetts: Addison-Wesley Publishing Company, 1986.

18. Data, C. J. and with a special contribution by Andrew Warden. *Relational Database Writings 1985-1989*. Reading Massachusetts: Addison-Wesley Publishing Company, 1990.

19. Data, C.J. *An Introduction to Database Systems*, vol. 1, The Systems Programming Series, 5 Edition. Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.

20. Dyer, Douglas E. and Mark A. Roth. "Object-Oriented Design Unifies Databases and Applications." Tech. Rep., , Wright-Patterson AFB, OH 45433, , AFIT/EN-TR-92-2, July 1992, Research funded by Joint Modeling and Simulation Systems Program Office, ASC/RWWW, Wright-Patterson AFB OH.

21. Dyer, Douglas E. "An Eclectic Method for Object-Oriented Database Design." Tech. Rep., , Wright-Patterson AFB, OH 45433, , AFIT/EN-TR-92-4, March 1992, Research paper for CSCE 746 Advanced Database Management Systems.

22. France, Robert B. "Semantically Extended Data Flow Diagrams: A Formal Specification Tool." *IEEE Transactions on Software Engineering*, vol. 18, no. 4, pp. 329-346, April 1992.

23. Guido, Dawn. *Class Notes on WCSE 473, Principles and Applications of Software Design: Object-Oriented Design.* 1992, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH.

24. Guido, Dawn and Michael Dedolph. "Object-Oriented vs 'Traditional' Software Development." in *The Fourth Annual Software Technology Conference: Joining Forces to Engineer Success (Tutorials).* Air Force Institute of Technology, April 1992, Sponsored by: HQ USAF/SC and USAF STSC.

25. Gupta, Rajiv and Ellis Horowitz. "A Guide to the OODB Landscape." in *Object-Oriented Databases with Application to CASE, Networks, and VLSI CAD*, Gupta, Rajiv and Ellis Horowitz, Eds. Englewood Cliffs, New Jersey: Prentice-Hall, 1991, chap. 1, pp. 1-11.

26. Harel, David. "Statecharts: A Visual Formalism for Complex Systems." *The Science of Computer Programming*, vol. 8, pp. 231-274, 1987.

27. Harel, David. "Biting the Silver Bullet: Toward a Brighter Future for System Development." *Computer*, vol. 26, no. 1, pp. pp. 8-20, January 1992.

28. Hartrum, Thomas C. *Class handout distributed in CSCE 594, Software Analysis and Design II.* Fall 1991, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH.

29. Hayes, Fiona and Derek Coleman. "Coherent Models for Object-Oriented Analysis." in *ACM OOPSLA'91 Conference Proceedings.* 1991, pp. 171-183.

30. *Holy Bible, New International Version, The.* International Bible Society, 1984.

31. Huges, John G. *Object-Oriented Databases*, Prentice Hall International Series in Computer Science. New York: Prentice Hall International, 1991.

32. Humphrey, Watts S. *Managing the Software Process*, The SEI Series in Software Engineering. Addison-Wesley Publishing Company, 1989.

33. Kim, Hyoung-Joo. "Algorithmic and Computational Aspects of OODB Schema Design." in *Object-Oriented Databases with Application to CASE, Networks, and VLSI CAD*, Gupta, Rajiv and Ellis Horowitz, Eds. Englewood Cliffs, New Jersey: Prentice-Hall, 1991, chap. 3, pp. 26-61.

34. Konstantinow, George. "Transition to Object-Oriented Development: Promoting a New Paradigm." in *Object-Oriented Databases with Application to CASE, Networks, and VLSI CAD*, Gupta, Rajiv and Ellis Horowitz, Eds. Englewood Cliffs, New Jersey: Prentice-Hall, 1991, chap. 6, pp. 92-100.

35. Korth, Henry F. and Abraham Silberschatz. *Database System Concepts*. New York: McGraw-Hill, 1991.

36. Levi, Shem-Tov and Ashok K. Agrawala. *Real-Time System Design*, McGraw-Hill Computer Science Series, and Series in Systems. McGraw-Hill Publishing Company, 1990.

37. Limbaugh, Rush H. III. *The Way Things Ought to Be*. New York: Pocket Books, 1992.

38. Lyngbaek, Peter. "Object-Oriented Database Systems." in *Sixth Annual OOPSLA Conference*. Phoenix, Arizona, October 1991, Tutorial 10 Notes.

39. Maier, David, Jacob Stein, Allen Otis, and Alan Purdy. "Development of an Object-Oriented DBMS." in *Research Foundations in Object-Oriented and Semantic Database Systems*, Cárdenas, Alfonso F. and Dennis McLeod, Eds. Englewood Cliffs, New Jersey: Prentice-Hall, 1990.

40. Mancl, Dennis and William Havanas. "A Study of the Impact of C++ on Software Maintenance." in *IEEE Conference on Software Maintenance - 1990*. The Institute of Electrical and Electronics Engineers, Inc., IEEE Computer Society Press, San Diego, California, November 1990.

41. Martin, James and James J. Odell. *Object-Oriented Analysis and Design*. Englewood Cliffs, New Jersey: Prentice-Hall, 1992.

42. Ozkarahan, Esen. *Database Management: Concepts, Design and Practice*. Englewood Cliffs, New Jersey: Prentice-Hall, 1990.

43. Painter, Michael K. "Information Integration for Concurrent Engineering (IICE): Program Foundations and Philosophy." April 1991, Armstrong Laboratory, Human Resources Directorate, Logistics Research Division, Wright-Patterson AFB OH.

44. Roth, Mark A. *Class lecture notes taken in CSCE 646, Computer Database Systems.* Fall 1991, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH.

45. Rumbaugh, James, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice-Hall, 1991.

46. Shlaer, Sally and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press Computing Series. Englewood Cliffs, New Jersey: Prentice-Hall, 1988.

47. Silberschatz, Avi, Michael Stonebraker, and Jeff Ullman. "Database Systems: Achievements and Opportunities." *Communications of the ACM*, vol. 34, no. 10, pp. pp. 110-120, October 1991.

48. Stonebraker, Michael, Lawrence A. Rowe, Bruce Lindsay, James Gray, Michael Carey, Michael Brodie, Philip Bernstein, and David Beech. "Third-Generation Data Base System Manifesto." in *Proceedings of the IFIP DS-4 Workshop on Object-Oriented Databases*. Windermere, England, July 1990.

49. Teorey, Toby J. *Database Modeling and Design: The Entity-Relationship Approach*. San Mateo, California: Morgan Kaufmann Publishers, 1990.

50. Vossen, Gottfried. *Data Models, Database Languages and Database Management Systems*, International Computer Science Series. Wokingham, England: Addison-Wesley Publishing Company, 1991.

51. Walker, Ian J. "Requirements of an Object-Oriented Design Method." *Software Engineering Journal*, pp. 102-113 , March 1992.

52. Waters, Sgt., Telephone interview. Air Force Standard Systems Center, 9 October 1992.

53. Zdonik, Stanley B. and David Maier. "Fundamentals of Object-Oriented Databases." in *Readings in Object-Oriented Database Systems*. San Mateo, California: Morgan Kaufmann Publishers, 1990.

Vita

Captain Kevin Joseph Routhier was born on January 1, 1962 in Lawrence, Massachusetts. He graduated from the Greater Lawrence Regional Vocational Technical High School in June 1979 and enlisted in the United States Air Force in August of the same year. He was first assigned as a student "voice processing specialist" at the Defense Language Institute Foreign Language Center where he graduated with honors from the Hungarian language department. From there at the Presidio of Monterey, California he was accepted to the United States Air Force Academy Preparatory School. Upon completion of preparatory school, in June 1981, he was appointed a cadet in the United States Air Force Academy itself.

On 29 May 1985, he graduated with a Bachelor of Science degree in Electrical Engineering and received a regular commission in the United States Air Force. After a year in pilot training at Vance AFB, he served his first "real-Air Force" tour at (what was then) Strategic Air Command Headquarters, Offutt AFB, Nebraska. There he advanced from computer systems analyst (programmer) to branch chief of maintenance for one of the Joint Strategic Target Planning Staff's major strategic war planning programs, in direct support of our nation's Single Integrated Operations Plan.

In May 1991 he entered the School of Engineering, Air Force Institute of Technology, as a student of Computer Systems' in both the Software Engineering and Database sequences. Upon graduation he was assigned to the 1881 Communications-Computer Systems Group at Hill AFB, Utah.

Permanent address: c/o Mr. Joseph N. L. Routhier
73 Inman Street
Lawrence, Massachusetts 01843

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE

ENTITY-RELATIONSHIP VERSUS OBJECT-ORIENTED MODELING AND THE UNDERLYING DBMS

5. FUNDING NUMBERS

6. AUTHOR(S)

Kevin J. Routhier, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Air Force Institute of Technology, WPAFB OH 45433-6583	AFIT/GCS/ENG/92D-15

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution unlimited	

13. ABSTRACT (*Maximum 200 words*)

Despite the impressive accomplishments in relational database research, greater support is needed for persistence of the new types of data encountered with object-oriented programming. The concept of object-orientation is not new in the realm of programming; however, its utilization in database management systems is still immature. Regardless of this fact, there is an urgency for object-oriented database technology.

With this increase in demand for the next generation databases comes the need to examine object-oriented data modeling versus the conventional entity-relationship modeling of relational database design. This thesis objective is to analyze both paradigms to determine if object-oriented modeling can significantly improve Department of Defense systems. After analyzing the entity-relationship paradigm and a representation of object-oriented modeling techniques we see a unifying of conceptual models encompassing both application and database development. Object-orientation's higher level of abstraction enables modeling of all problem domains and provides a common language between developer and client.

The critical issue in adoption of the object-oriented paradigm becomes not whether to adopt, but how to adopt object-oriented techniques. The benefits of object-oriented technology close the semantic gap by helping the computer to "see" things our way.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Object-Ortiented, Entity-Relationship, Modeling, Database, Object-Oriented Database, Relational Database, Analysis, Design	164
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL